

**ПРОФЕССИОНАЛЬНОЕ
ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ СИ**

А.И.Касаткин

**Системное
программирование**

А.И. Касаткин

**ПРОФЕССИОНАЛЬНОЕ
ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ СИ**

**СИСТЕМНОЕ
ПРОГРАММИРОВАНИЕ**

**Минск
"Вышэйшая школа"
1993**

ББК 32.973-01

K28

УДК 681.3.06

Касаткин А.И.

K28 **Профессиональное программирование на языке Си.**
Системное программирование . - Мн.: Выш. шк., 1993. -
301 с.: ил.

ISBN 5-339-00809-6.

Дается подробное описание практического использования новейших версий языка Си (Turbo C, Turbo C++, Borland C++). Рассматриваются интерфейс Си-функций и процедур на ассемблере, технология разработки резидентных программ, управление адаптерами асинхронной последовательной и параллельной связи, ввод информации с манипулятора "мышь", управление отображаемой (LIM EMS) и расширенной (LIM XMS) памятью.

Для пользователей ПЭВМ: студентов, аспирантов, программистов, инженеров, всех желающих составлять программы на Си.

2404010000-014

К 30-93
 M304(03)-93

ББК 32.973-01

ISBN 5-339-00809-6

© А.И. Касаткин, 1993

ПРЕДИСЛОВИЕ

Предлагаемая вниманию читателя книга завершает комплекс из трех книг под общим названием “Профессиональное программирование на языке Си”. Поскольку за основу приняты реализации языка программирования Си Borland International*, то везде, где упоминается язык программирования Си, имеется в виду как ANSI-стандарт, так и его расширения, реализованные в системах программирования Turbo C версии 2.01, Turbo C++ версии 1.01, Borland C++ версий 2.0 и 3.0. Ссылка на Turbo C означает общие, реализованные во всех версиях возможности.

Основное внимание в данной книге уделено наиболее сложным вопросам программирования: интерфейсу Си-программы с модулями, написанными на языке ассемблера, управлению адаптерами асинхронной последовательной и параллельной связи, вводу информации с манипулятора “мышь”. Достаточно подробно рассматривается дизайн резидентных программ, разработка которых представляет собой, пожалуй, наиболее трудную, не имеющую универсального решения задачу. Поскольку учебная литература, посвященная этим вопросам, полностью отсутствует, то изложение ведется в форме, удобной для обучения программистов и самообразования.

* Turbo C®, Turbo C++®, Borland C++®, Turbo Debugger®, Turbo Profiler® — зарегистрированные торговые марки Borland International; Intel® — зарегистрированная торговая марка Intel Corporation; IBM®, IBM XT/AT®, IBM PS/2® — зарегистрированные торговые марки International Business Machines Corporation; MS-DOS® — зарегистрированная торговая марка Microsoft Corporation.

Большая группа обсуждаемых в книге вопросов касается организации обмена информацией между двумя компьютерами с использованием адаптеров последовательной связи. Рассматриваются три возможные схемы подключения компьютеров и приводится вполне работоспособная программа переноса файлов.

В заключительной, шестой главе книги подробно рассматривается управление памятью персонального компьютера сверх лимита в 640К байт: отображаемой (LIM EMS) и расширенной (LIM XMS).

Приводимые примеры программ тщательно проверялись на совместимость в различных версиях продуктов Borland International и должны правильно работать в средах Turbo C, Turbo C++ и Borland C++. Однако в ходе подготовки книги к изданию вследствие сокращения объема пришлось отказаться от ряда комментариев в текстах программ и расположить их максимально компактно, что, естественно, ухудшило читабельность листингов. При этом в тексты программ могли вкрасться опечатки, за которые автор заранее приносит извинения читателям. При подготовке книги к изданию использовалась авторизованная копия системы программирования Borland C++ (номер пакета No DA142E10002171), являющаяся собственностью специального факультета переподготовки инженерных кадров Минского радиотехнического института.

Автор выражает благодарность А.Н. Вальвачеву за ряд ценных замечаний по содержанию и оформлению книги, а также А.А. Москалеву — за выполнение компьютерной верстки при подготовке оригинала-макета.

Оригинал-макет подготовлен автором, при этом сохранены авторский стиль изложения и оформления материала.

Все замечания и пожелания просьба присылать по адресу: 220048, Минск, проспект Машерова, 11, издательство "Вышэйшая школа".

Автор

1. ИНТЕРФЕЙС СИ-ПРОГРАММЫ С АССЕМБЛЕРОМ

1.1. Включение ассемблерных фрагментов в Си-функцию

Одна из часто встречающихся задач профессионального программирования на языке Си — это совместное использование модулей, написанных на Си, и модулей, написанных на ассемблере. В данной главе подробно рассматриваются существующие возможности совместного использования ассемблера и Turbo C. Усвоение материала данной главы не требует знания ассемблера, что для профессионального программиста является обязательным.

Наиболее простым способом объединения частей программы, написанных на Си и ассемблере, является использование в тексте Си-функции непосредственно ассемблерных инструкций или, как говорят, встроенного (in-line) ассемблера. Этот прием идеален для несложных ассемблерных процедур, таких как обращения к функциям BIOS или MS-DOS. Возможности Borland C++ по использованию встроенного ассемблера намного более широки по сравнению с Turbo C версии 2.0. Далее рассматривается использование встроенного ассемблера для Borland C++.

Любая строка Си-программы, начинающаяся с ключевого слова `asm`, до символа `';` или символа перевода строки рассматривается как инструкция на ассемблере. Для встроенного ассемблера разрешается использовать:

- 1) исполняемые инструкции (машинные команды);

2) цепочечные примитивы (инструкции манипуляции подряд расположенными байтами или словами) с префиксом повторения и без него ;

3) все инструкции передач управления (JUMP, CALL, LOOP, RET, IRET) с прямой или косвенной адресацией;

4) директивы описания и распределения данных (db, dw, dd, extrn).

Таким образом, встроенный ассемблер, как и следовало ожидать, не поддерживает использование директив ассемблера, управляющих его работой: ASSUME, SEGMENT, ENDS, PROC, ENDP, ORG и т.д. При задании операндов в ассемблерных инструкциях разрешается ссылаться на уже описанные переменные Си-функции, поля структурных переменных и объединений, метки, имена функций и т.п. В ассемблерных инструкциях можно использовать имя группы DGROUP, имена сегментов _TEXT, _STACK, _DATA и т.п. (см. вторую книгу комплекса, 6.3). Группа инструкций на ассемблере, заключенная в фигурные скобки, не требует повторения перед каждой из них ключевого слова asm, если используется компилятор Borland C++ версии 2.0 и старше.

В качестве примера рассмотрим функцию printche(), выводящую непосредственно в видеобуфер символ symb с заданным атрибутом attr n раз. Символы помещаются на экране, начиная со строки str и столбца col. Вариант этой функции на Си рассмотрен во второй книге комплекса (листинг L9_8.C). Левый верхний угол экрана соответствует координатам (0,0). В случае успеха функция возвращает константу ОК. Если заданы недопустимые str и stolb, а также в случае, когда запрошенное число n символов не уместится на экране, возвращается константа BAD_PARAM. Для вывода используется активная видеостраница адаптера. Символические константы описаны в заголовочном файле "screen.h", листинг которого приведен в 8.4 второй книги комплекса. Для повышения производительности функция использует следующие внешние переменные: vid_memory — начальный параграф видеопамати адаптера; start_adr — смещение до начала активной страницы; max_stolb — максимальное число

столбцов на экране (40, 80); max_str — максимальное число строк на экране (25, 28, 43, 50).

```
/* L1_1.C */
#pragma inline
#include <dos.h>
#include "screen.h"
printche(int str, int stolb, char symb, char atr, int n)
{
    extern unsigned vid_memory, start_adr, max_stolb,
                  max_str;
    /* Проверка параметров на допустимость. */
    if(str >= 0 && str <= max_str &&
        stolb >= 0 && stolb <= max_stolb)
    {
        asm {
            /* work = start_adr + ((max_stolb * str + stolb) << 1) */
            mov ax, max_stolb
            mul str
            add ax, stolb
            shl ax, 1;
            mov si, start_adr
            add si, ax
            /* word = (unsigned)(atr << 8) | symb; */
            mov bh, atr
            mov bl, symb
            /* old_ds = _DS; */
            /* _DS = vid_memory; */
            push ds
            mov cx, vid_memory
            mov ds, cx
            mov cx, n; подготовка счетчика цикла
            mov dx, 03dah
        }
        /* while((inportb(_DX) & 0x01)); */
wait1:
        asm {
            in     al, dx
            test   al, 01
            jne    wait1
        }
        /* while(!(inportb(_DX) & 0x01)); */
    }
}
```



```

wait2:
    asm {
        in     al,dx
        test   al,01
        je     wait2
        /* disable(); */
        cli
        /* * work = word; */
        mov     [si],bx
        /* enable(); */
        sti
        /* work ++; */
        inc     si
        inc     si
        /* while(n-) */
        dec     cx
        jne wait1:
        pop ds; /* восстановление ds */
    }
    return OK;
}
else
    return BAD_PARAM;
}

```

В приведенном листинге комментариями **/**/** отмечен эквивалент предложения из Си-функции, записываемый на ассемблере. В данном примере сохранены начало и конец функции, написанные на Си. Вся средняя часть переписана на ассемблере. Отметим, что встроенный ассемблер не допускает описания ассемблерных меток. Разрешается ссылка только на Си-метки, используемые в операторе **goto** языка Си. Поэтому метки **wait1** и **wait2** выведены за пределы блоков на ассемблере.

Встретив инструкции, помеченные ключевым словом **asm**, Borland C++ выполняет автоматический перезапуск компилятора с опцией **-B**, что означает генерацию ассемблерного кода, а затем вызов ассемблера **BASM.EXE** для получения объектного файла. Использование директивы препроцессора **#pragma inline** вызывает запуск компилятора сразу с опцией **-B**. Это сокращает общее время компиляции. Исполь-

зуя опцию -Efilename, можно вместо BASM.EXE подключить и другой ассемблер; его имя — filename.

Использование встроенного ассемблера в данном примере приводит к тому, что объектный модуль L1_1.OBJ по сравнению с L9_8.OBJ (см. вторую книгу комплекса) оказывается на несколько десятков байтов короче при приблизительно такой же производительности. Это подтверждает известный факт, что программирование только на ассемблере по сравнению с использованием Си дает код, который короче приблизительно на 20% и производительнее на 5%.

Использование встроенного ассемблера на Turbo C версии 2.0 не позволяет компилировать программу в интегрированной среде, а требует обращения к компилятору командной строки TCC.EXE. Кроме того, не разрешается описание сразу группы ассемблерных инструкций, заключенных в блок { }, перед началом которого стоит единственное ключевое слово asm. При работе в Turbo C версии 2.0 ключевое слово asm приходится повторять перед каждой строкой встроенного ассемблера.

Еще одной возможностью помещения машинного кода непосредственно в Си-функцию является использование библиотечной функции `__emit__()`, прототип которой помещен в файле `<dos.h>`.

```
#include <dos.h>
```

```
void __emit__(argument, ...)
```

Помещает в кодовый сегмент программы байты, записанные через запятую в списке аргументов.

Например, приведенное ниже обращение к функции `__emit__()` поместит в кодовый сегмент “фирменный определитель” функции “KASATKIN A.I.”. Так как байты символов не должны рассматриваться как код, фрагмент, вставляемый в кодовый сегмент, начинается с инструкции безусловной передачи управления на байт кода, расположенный сразу после текста. Определитель всегда позволит “узнать” свои программы, даже если они и будут распространяться под другим именем. Для этого достаточно выполнить дамп кодового сегмента .EXE-файла в любом

из отладчиков, например TD.EXE (подробнее о турбо-отладчике см. в первой книге комплекса):

```
...
__emit__(0xEB, 0x0D, 'K', 'A', 'S', 'A', 'T', 'K', 'I',
          'N', ' ', 'A', '.', 'I', '.');
...
```

Приведенный фрагмент эквивалентен следующему фрагменту на ассемблере:

```
...
    asm JMP m1
    asm db 'KASATKIN A.I.'
    m1:
...
```

Использование встроенного ассемблера в функциях-членах классов имеет некоторые особенности организации доступа к данным. (Подробнее о классах см. в первой книге комплекса.)

Напомним, что при вызове функции-члена класса ей передается неявный параметр `this`, равный указателю на объект, для которого эта функция вызвана. Для доступа к данным следует использовать ссылку, сконструированную из `this` и имени переменной, соединенных символом `'.'`. С “точки зрения” ассемблера члены класса представляют собой поля структуры и для доступа к ним требуется предварительная установка регистра `BX` (для моделей памяти, в которых указатель на данные по умолчанию `near`) или `ES` и `BX` (для моделей памяти, в которых указатель на данные по умолчанию `far`). Далее приводится пример переработанной программы `L12_5.CPP`, рассмотренной в первой книге комплекса. Использование встроенного ассемблера демонстрирует функция `int fasm(int)`. Приводимый пример рассчитан на компиляцию в моделях памяти `MEDIUM`, `LARGE` или `HUGE`:

```
// L1_2.CPP
#include <iostream.h>
class my_date {
```



```

    int day, month, year;
public:
    // Различные варианты конструкторов:
    my_date();    // дата по умолчанию – используется
                  // текущая дата
    my_date(char * str); // дата в формате "ДДММГГГГ"
    my_date(char * d, char * m, char * y); // дата в
                  // формате "День", "Месяц", "Год"
    void print_date(void); // вывод даты в удобочитаемом
                          // формате
    int asmf(int offset); // функция со встроенным
                          // ассемблером
};

// Функция возвращает значение поля данных, смещенное
// в памяти относительно this->day на ofs байтов.
int my_date::asmf(int ofs){
    asm les bx, this // для FAR-указателей
    asm add bx, ofs
    asm mov ax, es:[bx].day
    return _AX;
}

void my_date::print_date(void)
{
    // Печать даты в формате ДД.ММ.ГГГГ
    cout << day << '.' << month << '.' << year << '\n';
}
#include <dos.h>
my_date::my_date()
{
    struct date cur_date;
    getdate(&cur_date);
    day    = cur_date.da_day;
    month  = cur_date.da_mon;
    year   = cur_date.da_year;
}
#include <stdlib.h>
my_date::my_date(char * d, char * m, char * y)
{
    day    = atoi(d);
    month  = atoi(m);

```



```

    year = atoi(y);
}
my_date::my_date(char * str)
{
    year = atoi(str+4); * (str+4) = '\0';
    month = atoi(str+2); * (str+2) = '\0';
    day = atoi(str);
}
void main(void)
{
    my_date one_date = my_date();
    one_date.print_date();
    cout << one_date.asmf(0) << '\n';
    my_date sec_date = my_date("17", "05", "1954");
    sec_date.print_date();
    cout << sec_date.asmf(2) << '\n';
    my_date thr_date = my_date("17051954");
    thr_date.print_date();
    cout << thr_date.asmf(4) << '\n';
}

```

В программе с использованием перегружаемых конструкторов создаются и инициализируются три различных объекта класса `my_date`: `one_date`, `sec_date` и `thr_date`. Для каждого из них распечатываются значения данных объекта и возвращаемое функцией `asmf()` целое число. Оно представляет собой содержимое памяти по смещению от указателя `this -> day` соответственно на 0, 2 и 4 байта. Далее приводятся результаты исполнения программы:

```

G:\>L1_2.exe
3.5.1992          — соответствует текущей дате компиляции
3                — значение one_date.day
17.5.1954
5                — значение sec_date.month
17.5.1954
1954             — значение thr_date.year

```

Для программы, компилирующейся с моделями памяти TINY, SMALL и COMPACT, функция `asmf()` должна быть переработана следующим образом:


```
// Функция возвращает значение поля данных, смещенное
// в памяти относительно this на ofs байтов.
int my_date::asmf(int ofs){
    asm mov bx, this // для NEAR-указателей
    asm add bx, ofs
    asm mov ax, [bx].day
    return _AX;
}
```

Используя директивы условной компиляции, несложно разработать функцию, рассчитанную на любую модель памяти.

1.2. Вызов ассемблерной процедуры из Си-функции

В данном параграфе рассматривается общая схема построения ассемблерной процедуры, которая вызывается из Си-функции с передачей параметров и возвращает в точку вызова значение того или иного типа. Предполагается, что:

- 1) ассемблерные процедуры и Си-функции компилируются совместно; при этом используется файл проекта, в котором перечисляются Си-функции и объектные модули, полученные ассемблированием функций на языке ассемблера с помощью TASM.EXE (для Turbo C версии 2.01); если используется IDE Borland C++, ассемблер для получения объектного файла вызывается автоматически и можно в файл проекта включить ассемблерную процедуру (файл с расширением .ASM);

- 2) функция main() составлена на языке Си.

Ассемблерная процедура должна удовлетворять дополнительным требованиям, позволяющим:

- 1) гарантировать получение всей необходимой информации редактором связей;

2) обеспечить получение значений аргументов, переданных ей при вызове, и видимость всех внешних переменных;

3) гарантировать соответствие моделей памяти, принятых при компиляции Си-функций и при обработке ассемблерных процедур.

В этой связи ассемблерная процедура должна:

1) использовать правила именования сегментов, принятые в Turbo C;

2) явно описывать все глобальные и внешние идентификаторы;

3) поддерживать принятую в Си последовательность передачи параметров и возврата значений в точку вызова.

Правила именования сегментов, принятые в Turbo C, приведены во второй книге комплекса (см. табл. 6.2). Обычно модули на ассемблере состоят из трех секций: кода, инициализированных и неинициализированных данных. Секция стека необязательна, так как она автоматически будет создаваться программой-загрузчиком системы программирования (файл C0x.OBJ).

Турбо-ассемблер TASM.EXE версии 2.0 и старше (для Turbo C++ и Borland C++), макро-ассемблер MASM 5.0 и старше предусматривают директивы упрощенного описания модели памяти и сегментов:

```
.MODEL { SMALL | COMPACT | MEDIUM | LARGE | HUGE }
```

.CODE — описание сегмента кода;

.DATA — описание сегмента инициализированных данных;

.DATA? — описание сегмента неинициализированных данных.

Ассемблеры TASM.EXE и BASM.EXE дополнительно поддерживают и модель памяти TINY, которая, в принципе, равносильна модели SMALL.

В общем случае процедура на ассемблере должна строиться по такой схеме:


```

.MODEL SMALL
.DATA
    ...
    сегмент инициализируемых данных процедуры
    ...
.DATA?
    ...
    сегмент неинициализируемых данных процедуры
    ...
.CODE
    ...
    кодовый сегмент процедуры без директив описания
сегментов
    ...
END; директива завершения модуля

```

При обработке приведенных выше директив ассемблер выполнит их замещение, порождая следующий ассемблерный модуль:

```

DGROUP GROUP _DATA, _BSS ; для HUGE – отсутствует
data SEGMENT WORD PUBLIC 'DATA'
; для HUGE Turbo C++ и Borland C++
; data SEGMENT WORD PUBLIC 'FAR_DATA'
    ...
    сегмент инициализируемых данных процедуры
    ...
data ENDS
_BSS SEGMENT WORD PUBLIC 'BSS'
    ...
    сегмент неинициализируемых данных процедуры
    ...
_BSS ENDS
code SEGMENT BYTE PUBLIC 'CODE'
    ASSUME CS:code, DS:DGROUP;
; для HUGE Borland C++
; ASSUME CS:code, DS:data
    ...
кодовый сегмент процедуры без директив описания сегментов
    ...
code ENDS
END; директива завершения модуля

```


Если необходимо (желательно) отказаться от директив описания сегментов, можно непосредственно оформить ассемблерный модуль по приведенной схеме.

Имена code, data имеют значения, зависящие от модели памяти (табл. 1.1).

Табл. 1.1. Правила замещения идентификаторов code и data для различных моделей памяти

Модель памяти	Имя сегмента	Размер указателя и адресация	
		код	данные
TINY, SMALL	code = _TEXT data = _DATA	DW _TEXT:xxxx	DW DGROUP:xxxx
COMPACT	code = _TEXT data = _DATA	DW _TEXT:xxxx	DD DGROUP:xxxx
MEDIUM	code = filename_TEXT data = _DATA	DD filename_TEXT:xxxx	DW DGROUP:xxxx
LARGE	code = filename_TEXT data = _DATA	DD CS:xxxx	DD DGROUP:xxxx
HUGE	code = filename_TEXT data = filename_DATA	DD filename_TEXT:xxxx	DD filename_DATA:xxxx

П р и м е ч а н и е . xxxx - значение смещения до адресуемой памяти; filename - имя файла, содержащего компилируемую функцию.

Все внешние переменные, описанные в Си-функциях как внешние, на которые ссылается ассемблерная процедура, должны описываться в ассемблерном блоке явно с использованием директивы EXTRN:

EXTRN имя_переменной: размер

где размер задает число байтов, выделяемых компилятором Си, и равен: BYTE для объектов типа char; WORD для объектов типа int, unsigned, near-указателей; DWORD для объектов типа long, float, far-указателей; QWORD для объектов типа double (8 байт); TBYTE для объектов типа long double (10 байт).

Если имеются переменные, описанные в пределах ассемблерного модуля, на которые выполняется ссылка в Си-функции, то они объявляются как PUBLIC:

PUBLIC имя_переменной

Имя ассемблерной процедуры, вызываемой из Си, в ассемблерном блоке должно описываться как PUBLIC, что делает это имя видимым при редактировании связей.

Turbo C накладывает определенные ограничения на запись имен переменных и функций, вытекающие из особенностей работы компилятора и компоновщика.

1. При объявлении внешнего идентификатора, в том числе и имен функций, которые по умолчанию являются внешними, компилятор Си автоматически добавляет символ подчеркивания '_' перед именем и в таком виде сохраняет его в объектном модуле. Подобное поведение соответствует включенной опции интегрированной среды Main Menu->Options->Compile->Code Generation->Generate Underbars (или опции компилятора командной строки -u, выбираемой по умолчанию). Если данная опция интегрированной среды выключена (опция компилятора командной строки -u-), символ подчеркивания перед идентификатором не добавляется и все идентификаторы Си совпадают с идентификаторами в ассемблерном модуле. В то же время ассемблеры TASM и MASM не добавляют символ подчеркивания. Таким образом, внешние идентификаторы в ассемблерном модуле должны иметь лидирующий символ подчеркивания, если Си-модуль компилируется с добавлением подчеркивания, и не должны содержать его в противном случае. Еще раз подчеркнем, что все сказанное относится только к внешним идентификаторам. Внутренние переменные процедуры на ассемблере могут иметь любые имена.

2. Компилятор Turbo C для всех внешних идентификаторов сохраняет регистр букв, т.е. является "регистрочувствительным" (Case Sensitive), и такое поведение Си-компилятора невозможно изменить. В то же время ассемблеры TASM.EXE, TASM.EXE и MASM.EXE при за-

писи внешних идентификаторов в объектный модуль приводят их к верхнему регистру, т.е. все малые буквы преобразуют в большие. Такое поведение по умолчанию можно переопределить, используя опцию **/mx**. Ассемблирование с этой опцией сохраняет регистр букв глобальных идентификаторов.

Последнее, что необходимо учитывать при разработке процедуры на ассемблере, вызываемой из Си-функции, относится к передаче параметров через стек и возврату значений в точку вызова. Рассмотрим, например, вызов из функции `main()` некоторой функции `func()`:

```
void func(int arg1, int arg2, int arg3); /* прототип
                                           функции */
main()
{
    int arg1 = 1, arg2 = 5, arg3 = 4;
    func(arg1, arg2, arg3); /* вызов функции */
    ...
}
```

Когда функция вызывается из Си-модуля, она получает стек в состоянии, которое иллюстрирует рис. 1.1.

Таким образом, регистр BP указывает на последний параметр, передаваемый в стек. Готовясь к вызову функции, точка вызова “пуширует” в стек копии аргументов. Аргументы “пушируются” в стек справа налево. В результате они располагаются в стеке так, как показано на рис. 1.1: первый аргумент имеет адрес, меньший, чем второй, второй — меньший, чем третий, и т.п. Разность между BP и SP часто называют *фреймом стека*.

Важно, чтобы вызываемая процедура перед возвращением в точку вызова восстановила все изменяемые регистры, за исключением тех, которые используются для передачи возвращаемого значения (AX при возврате слова или AX и DX при возврате двойного слова). Это относится и к значениям SP и BP. Содержимое фрейма стека также следует оставить неизменным (по крайней мере, адрес точки возврата). Поэтому рекомендуемой практикой получения значений параметров из стека является

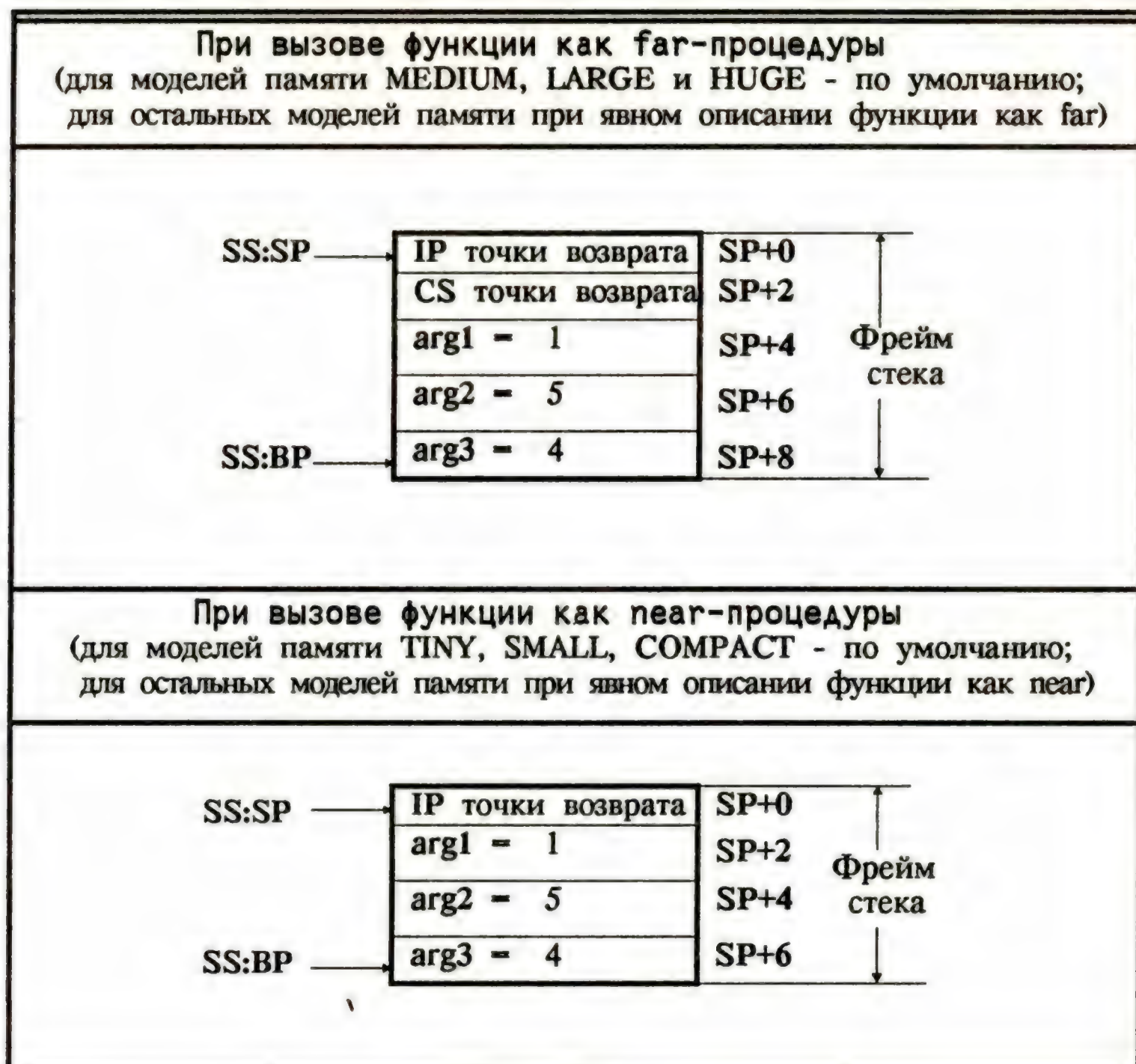


Рис.1.1. Стек в момент входа в функцию, вызванную из Си-модуля

не инструкция POP, а доступ к стеку с использованием косвенной адресации. Напомним, что любая ссылка на память типа [BP] предполагает использование в качестве сегментного регистра SS, а в качестве смещения — содержимого регистра BP. Стандартная для всех Си-функций процедура использования регистра BP заключается в следующем. Получив управление, функция сохраняет значения BP в стеке, а затем приравнивает BP текущему значению SP. Перед завершением процедуры BP восстанавливается. Другими словами, ассемблерный код Си-функции выглядит так:


```
PUSH BP      ; сохраняет в стеке текущее значение BP
MOV BP, SP   ; устанавливает BP равным текущей
              ; вершине стека
```

...

любые инструкции

...

```
POP BP       ; восстанавливает BP
RET          ; возврат из процедуры
```

Выполнение двух начальных инструкций позволяет впоследствии ссылаться на первый аргумент в стеке как на [BP+4] для функций, вызываемых как near-процедуры ([BP+6] для far-процедур). (Добавление 4(6 для far-процедур) к BP следует из того, что 2(4 для far-процедур) байта занимают адрес точки возврата, а еще 2 байта добавляются после выполнения инструкции PUSH, автоматически уменьшающей текущее значение SP.) Второй аргумент в стеке доступен по ссылке [BP+6] ([BP+8] для far-процедур) и т.д. Конечно, программист, составляющий процедуру на ассемблере, может и не использовать применяемый Си-компилятором прием. Однако автор рекомендует использовать именно этот способ.

Если процедура на ассемблере имеет возвращаемое значение, следует соблюдать правила, принятые компилятором Си. Если возвращается слово (char, short, int, enum и near-указатели), компилятор ожидает, что они помещаются в регистре AX (для char — в регистре AL). Если возвращаемое значение занимает 32 бита (2 слова), в AX помещается младшее слово, в DX — старшее. Значения типа float, double и long double возвращаются во внутреннем регистре сопроцессора 80x87, соответствующем текущей вершине стека сопроцессора (регистр ST(0)). При эмуляции сопроцессора возвращаемое значение помещается в регистр, который эмулирует ST(0). Вызывающая функция затем копирует возвращаемое значение тогда, когда оно требуется.

Структурные переменные длиной 1 байт возвращаются в AL, длиной 2 байта — в AX, а длиной 4 байта — в регистрах DX:AX. Если длина структурной переменной равна 3 байтам или больше 5 байт, возвращаемое зна-

чение записывается в статическую память и возвращается указатель на нее (AX — для младших моделей памяти, DX:AX — для старших моделей).

Далее приводится пример ассемблерной процедуры, корректно вызываемой из Си-функции. Процедура выполняет действия, аналогичные функции L8_11.C, листинг которой приведен в 8.6 второй книги комплекса: позиционирует курсор, обновляя область данных BIOSa в строку str, столбец stolb на странице видеопамати page. Позиция курсора во внутренних регистрах контроллера ЭЛТ не обновляется. В случае, когда запрошенное позиционирование выполнить невозможно, функция возвращает 255; в случае успеха возвращается 0. В Си-функции, вызывающей рассматриваемую функцию goto_xy(), описывается прототип

```
int goto_xy(int, int, int);
```

а сама функция вызывается так:

```
. . .  
goto_xy(stroka, stolb, page);  
. . .
```

Рассматриваемый ниже пример приведен для Си-функций, компилируемых с использованием моделей памяти TINY, SMALL, COMPACT, и предполагает использование TASM.EXE версии 2.0 или MASM версии 5.0 и старше.

```
; L1_3.ASM  
.MODEL SMALL  
.CODE  
  
PUBLIC _goto_xy; делает имя видимым из Си  
_goto_xy PROC NEAR  
_stroka EQU WORD PTR [bp+4]; аналогично #define в Си  
_stolb EQU WORD PTR [bp+6]; для внутренних переменных  
_page EQU WORD PTR [bp+8]; символ '_' не обязателен  
BAD_PARAM EQU 255  
OK EQU 0
```



```

    push bp      ; не обязательно, но очень удобно
    mov bp, sp  ; использовать Си-начало функции
    push ds      ; тело процедуры
    mov ax, 40h
    mov ds, ax   ; DS = 40h
; Проверка параметров на допустимость. Максимальное число
; столбцов определяется по области данных BIOS, адрес
; 40:4Ah,
; максимальное число строк – по адресу 40:84h (но не CGA!).
    mov ax, ds:[4ah]; в ax -> max_stolb
    cmp ax, _stolb  ; _stolb <= max_stolb ?
    jb _error      ; нет, завершение процедуры
    mov ax, ds:[84h]; в ax -> max_str
    jne not_cga    ; для CGA здесь будет нуль
    mov ax, 24     ; для CGA max_str = 24
not_cga:    cmp ax, _stroka ; _stroka <= max_str ?
    jb _error      ; нет, завершение процедуры
; Сформируем слово новой позиции для записи в область
; данных BIOS.
; cur_pos = (stroka << 8) | stolb;
    push cx
    mov ax, _stroka
    mov cl, 8
    shl ax, cl
    xor ax, _stolb
; Формируем смещение до нужного слова области данных BIOS,
; хранящей позицию курсора, и записываем ее.
; * (unsigned near *) (0x50 + (page << 1)) = cur_pos;
    push bx
    mov bx, _page
    shl bx, 1
    add bx, 50h
    mov [bx], ax
    pop bx      ; сохранение регистров – правильный
    pop cx      ; стиль программирования
    xor ax, ax   ; возврат OK из процедуры
    jmp short _exit
_error:    xor ah, ah
    mov al, BAD_PARAM ; возврат BAD_PARAM
_exit:    pop ds   ; сохранение регистров – правильный
    pop bp      ; стиль программирования

```



```

        ret          ; возврат в Си-функцию
_goto_xy ENDP
        END

```

Процедура на ассемблере порождает объектный файл, который на 15% короче своего эквивалента, полученного из функции L8_11.C (см. вторую книгу комплекса) при несколько более высокой производительности. Процедура должна ассемблироваться с опцией /mx. При отсутствии TASM версии 2.0 или MASM версии 5.0 и старше следует удалить из листинга L1_3.ASM директиву .CODE и оформить ассемблерную процедуру по такой схеме:

```

_TEXT      SEGMENT BYTE PUBLIC 'CODE' ; вместо
директивы .CODE
        ASSUME    CS:_TEXT, DS:DGROUP
        PUBLIC   _goto_xy
        . . . . .
кодový сегмент процедуры без директив описания сегментов
        . . . . .
_goto_xy   ENDP
_TEXT      ENDS
        END

```

Для получения объектного модуля, корректно работающего с Си-функцией, скомпилированной с моделями памяти MEDIUM, LARGE и HUGE, потребуется внести некоторые изменения в L1_3.ASM. Во-первых, директива .MODEL должна указывать нужную модель памяти. Во-вторых, директивы EQU для параметров _stroka, _stolb и _page должны записываться так:

```

_stroka EQU WORD PTR [bp+6 ]; EQU аналогично #define в Си.
_stolb  EQU WORD PTR [bp+8 ]; Для внутренних переменных
_page   EQU WORD PTR [bp+10]; символ '_' не обязателен.

```

В-третьих, следует определить _goto_xy как FAR-процедуру:

```

_goto_xy  PROC FAR

```


Рассмотренное поведение Си-компилятора соответствует так называемому *Си-порядку именования и передачи параметров*. Специальная опция интегрированной среды Main Menu->Options->Code Generation-> Calling convention устанавливает принимаемый по умолчанию порядок: либо Си, либо Паскаль. При использовании компилятора командной строки по умолчанию принимается Си-порядок. Опция -p устанавливает по умолчанию порядок именования переменных и передачи параметров, принятый в языке Паскаль (паскалевский порядок). В соответствии с ним Си-компилятор не добавляет символ подчеркивания при записи имен в объектный файл даже в том случае, когда опция компилятора Generate Underbars включена. Поэтому, если требуется подавить генерацию символа подчеркивания только для некоторых внешних идентификаторов, при их описании следует использовать модификатор pascal (в предположении, что по умолчанию принимается Си-порядок). Использование паскалевского порядка заставляет Си-компилятор преобразовывать имена к верхнему регистру букв. Преобразование выполняется либо для всех идентификаторов, если по умолчанию принят паскалевский порядок и идентификаторы не содержат модификатор cdecl, либо действует на идентификаторы, описанные явно как pascal, если по умолчанию установлен Си-порядок.

Описание функции с модификатором pascal воздействует не только на генерацию записываемого в .OBJ-файл имени, но и изменяет порядок помещения параметров в стек при вызове этой функции. Паскалевский порядок передачи параметров предполагает, что фактические параметры “пушируются” в стек не справа налево, а слева направо. В результате параметры в стеке располагаются в порядке, обратном приведенному на рис. 1.1. И последнее, на что влияет модификатор pascal. После каждого обращения к функции, описанной с модификатором cdecl, Си-компилятор увеличивает SP на размер фрейма стека. Так, например, после вызова функции goto_xy() будет записана инструкция add sp, 6. Для функций с модификатором pascal компилятор этого не делает. Ответствен-

ность за увеличение стека лежит на вызываемой функции. Как результат, функции, удовлетворяющие паскалевскому порядку, завершаются не обычной инструкцией `ret`, а инструкцией `ret n`, где `n` — размер стека фрейма. Поэтому, например, если бы вызываемая из Си-функции функция `goto_xy()` имела бы прототип

```
int pascal goto_xy(int, int, int);
```

либо по умолчанию был установлен паскалевский порядок именования и передачи параметров, следовало бы сделать в файле `L1_3.ASM` такие изменения:

1) заменить директивы `EQU` при определении `_stroka`, `stolb` и `page`, так как копии аргументов помещены в стек в обратном порядке:

```
...
_stroka EQU WORD PTR [bp+8] ; EQU аналогично #define в Си.
_stolb   EQU WORD PTR [bp+6] ; Для локальных переменных
_page    EQU WORD PTR [bp+4] ; символ '_' не является
                               ; обязательным.
...
```

2) заменить имя процедуры всюду, где оно встречается (убрать символ подчеркивания и записать имя большими буквами):

```
...
                PUBLIC GOTO_XY; делает имя видимым из Си
GOTO_XY         PROC NEAR
...
GOTO_XY         ENDP
...
```

3) заменить инструкцию `ret` на `ret 6`.

Разработка и отладка функции `L1_3.ASM` с принятым в языке Паскаль порядком передачи и возврата параметров послужат хорошим упражнением для самостоятельной работы. Следует отметить, что паскалевский порядок при-

нят за основу во многих библиотеках и средах, например в PARADOX ENGINE, WINDOWS и др.

При разработке ассемблерных процедур, являющихся функциями-членами класса, следует иметь ввиду, что первым параметром среди передаваемых через стек компилятор поместит указатель `this`. Его размер соответствует размеру указателя на данные для модели памяти, с использованием которой компилировалась вызывающая ассемблерную процедуру функция (для моделей памяти TINY, SMALL и MEDIUM — 2 байта; для остальных — 4 байта). Использование указателя `this` в ассемблерных инструкциях для доступа к данным объекта рассмотрено в 1.1.

1.3. Вызов Си-функции из ассемблерной процедуры

Необходимость вызова Си-функций из ассемблерной процедуры, как правило, возникает при использовании библиотечных функций Turbo C, выполняющих ввод-вывод или входящих в состав специализированных библиотек, таких, например, как HLEV.LIB для эффективного графического программирования или PARADOX ENGINE для эффективной разработки баз данных.

При вызове из ассемблерной процедуры библиотечных функций Turbo C следует иметь в виду одно важное обстоятельство. Как правило, эти функции ссылаются на внешние переменные, инициализируемые программой-загрузчиком C0x.OBJ, или предполагают наличие тех или иных условий, создаваемых процедурами программы-загрузчика: состояние сопроцессора 80x87, режим работы видеоадаптера, состояние памяти и т.п. Выполнение всех этих действий средствами ассемблерной процедуры вряд ли оправдано. Поэтому рекомендуемым способом построения ассемблерных процедур, вызывающих библиотечные функции, является следующий прием. На Си разрабатывается головная функция `main()`, содержащая единствен-

ную строку — вызов ассемблерной процедуры. Она ассемблируется в .OBJ-файл, подключаемый к головному модулю через файл проекта. После этого ассемблерная процедура может вызывать любую библиотечную функцию Turbo C или Си-функцию, в составе которой имеются обращения к библиотечным функциям. Именно такое построение программы предполагается при дальнейшем рассмотрении материала в данном параграфе.

Так как ассемблерная процедура будет вызываться из Си-функции, она должна удовлетворять общей схеме, рассмотренной в 1.2. Кроме того, ассемблерная процедура, вызывающая Си-функцию, должна удовлетворять некоторым дополнительным требованиям.

Имя вызываемой Си-функции должно быть описано как near- или far-указатели на функции директивой

```
EXTRN имя_функции: { FAR | NEAR }
```

Тип указателя должен соответствовать модели памяти, с использованием которой компилировалась Си-функция. Например, если планируется использовать в ассемблерной процедуре функцию printf() из библиотеки функций для моделей памяти TINY, SMALL или COMPACT, в ассемблерный модуль помещается строка

```
EXTRN _printf : NEAR
```

Имя функции записывается с лидирующим символом подчеркивания строчными буквами. Сказанное относится к библиотечным функциям Turbo C. Однако для других библиотек, использующих принятую в языке Паскаль последовательность передачи параметров и возврата значений, это может и не требоваться.

Далее приводится пример Си-функции, вызывающей ассемблерную процедуру demo_asm(), и текст самой ассемблерной процедуры. Ей передаются два параметра через стек: param1, param2, а также far-указатель stack_ptr на переменную типа int. Вызывая функцию Turbo C printf(), процедура выводит переданные ей через стек аргументы и

значения внешних переменных ext1 и ext2. Они описаны и инициализированы в Си-функции, вызывающей demo_asm(). Затем ассемблерная процедура demo_asm() записывает по адресу, на который указывает stack_ptr, константу, равную 20, и возвращает в точку вызова значение far-указателя на инициализированную внутри процедуры ASCIIZ-строку символов. Си-функция распечатывает возвращенное ассемблерной процедурой значение указателя и строку, на которую он указывает, а также содержимое ячейки памяти, на которую указывает stack_ptr. Следовательно, для связи с ассемблерной процедурой будут использованы как передача параметров через стек (Call-by-Value), так и передача значений через указатели (Call-by-Reference). По умолчанию установлен Си-порядок именования и передачи параметров. Приводимый пример рассчитан на компиляцию в моделях памяти MEDIUM, LARGE или HUGE.

```
/* L1_4.C */
#include <stdio.h>
char far * demo_asm(int, int, int far *); /*прототип
                                           функции */
int ext1 = 25;      /*эти внешние переменные будут */
long ext2 = 60000; /*видимы в ассемблерной процедуре */
void main(void)
{
    int local, param1 = 1024, param2 = 4096;
    int far * stack_ptr = (int far *)&local;
    printf("Строка, возвращенная ассемблерной",
           "процедурой -%s\n",
           demo_asm(param1, param2, stack_ptr));
    printf("Значение, переданное через указатель - %d\n",
           local);
}
```

Си-функция L1_4.C включается в файл проекта вместе с объектным файлом L1_5.OBJ, полученным ассемблированием приводимого далее ассемблерного кода (используется TASM версии 2.0). Ассемблерная процедура вызывает из своих пределов библиотечную функцию Turbo C printf() для вывода на печать двух целых аргументов,

переданных через стек при вызове из Си-функции, и значений внешних переменных `int ext1`, `long ext2`, описываемых в точке вызова. Возвращает `far`-указатель на строку символов, описанную и инициализированную ассемблерной процедурой.

```
; L1_5.ASM
L1_5_TEXT SEGMENT BYTE PUBLIC 'CODE'
DGROUP GROUP _DATA
L1_5_TEXT ENDS
_DATA SEGMENT WORD PUBLIC 'DATA'
FormPtr db 'Значения, переданные в demo_asm через стек:'
        db ' param1 = %d; param2 = %d', 0dh, 0ah; эквивалент '\n'
        db 'Внешние переменные: ext1 = %d; ext2 = %ld',
        0dh, 0ah, 0
AsmStr db 'Строка, описанная внутри demo_asm', 0
_DATA ENDS
```

```
        EXTRN _ext1:word, _ext2:dword; ссылка на имена Си
        PUBLIC _demo_asm           ; делает видимым из Си
        EXTRN _printf:far          ; ссылка на функцию
.MODEL LARGE
.CODE
_demo_asm PROC FAR
asm_parm1 EQU WORD PTR [bp+6 ]; первый параметр в стеке
asm_parm2 EQU WORD PTR [bp+8 ]; второй параметр в стеке
ptr_ofs   EQU WORD PTR [bp+10]; смещение far-указателя
ptr_seg   EQU WORD PTR [bp+12]; сегмент far-указателя
        push bp           ; не обязательно, но удобно
        mov bp, sp        ; использовать такое начало
        push bx
; Подготовка вызова библиотечной функции печати
        ; printf(FormPtr, asm_parm1, asm_parm2, ext1, ext2)
        mov bx, offset DGROUP:_ext2
        push [bx+2] ; в стек "пушируются" старшее
        push [bx]   ; и младшее слово переменной
                    ; типа long
        push DGROUP:_ext1
        push asm_parm2
        push asm_parm1
; В printf передается far-указатель на строку формата.
        mov ax, seg FormPtr
```



```

        push ax
        mov ax, offset FormPtr; Для SMALL и MEDIUM в
        push ax                ; стек не следует запи-
                                ; сывать seg FormPtr
        call _printf ; вызов библиотечной функции
        add sp, 14      ; стек "опускается" на размер
                        ; фрейма
; Запись в ячейку памяти с использованием far-указателя
; константы, равной 20
        push es
        mov es, ptr_seg
        mov bx, ptr_ofs
        mov ax, 0014h
        mov es:[bx], ax ; записана константа 20
        pop es ; восстанавливается измененный регистр
; Подготовка возвращаемого значения
        mov dx, seg AsmStr
        lea ax, AsmStr
        pop bx
        pop bp
        ret                ; возврат в Си-функцию
_demo_asm ENDP
        END

```

Модель памяти LARGE в процедуре demo_asm выбрана только для того, чтобы продемонстрировать интерфейс для такой модели. Использование старших моделей памяти при объединении модулей на Си и ассемблере часто порождает ошибку компоновщика "Fixup overflow". Ее причиной, чаще всего, является вызов функции из другого сегмента машинной инструкцией CALL NEAR или ссылка на данные в другом сегменте. Вызывать это могут самые неожиданные неточности программы. Рекомендуется следовать приведенной в L1_5.ASM схеме, не полагаясь на директиву .DATA при описании группы DGROUP.

2. РАЗРАБОТКА РЕЗИДЕНТНЫХ ПРОГРАММ

2.1. Введение в резидентные программы

2.1.1. Нерезидентное и резидентное завершение программы. Резидентная программа как обработчик прерывания

Во второй книге комплекса (см. 1.3) подробно описан механизм прерываний, реализованный персональными компьютерами. В данной главе рассматриваются основные приемы построения резидентных программ. Отличительное свойство этих программ состоит в том, что они после своего завершения остаются в памяти компьютера, а операционная система “защищает” занятую ими память от повторного использования. В литературе такие программы чаще всего называют TSR (Terminate-but-Stay-Resident), а иногда — “всплывающими” (Pop-Up) программами. Использование TSR позволяет расширить возможности системы по обслуживанию внешних устройств или реализовать так называемое “пассивное” мультипрограммирование. MS-DOS является однопрограммной системой, но активизация TSR вызывает “переключение” компьютера на резидентную программу. А если активизация TSR выполняется периодически, появляется возможность выполнения программ на фоне других программ — исполнение музыки, печать файлов, обслуживание связи между компьютерами и т.д. Неприспособленность MS-DOS к многопрограммной работе обуславливает особые требования к TSR, подробно рассматриваемые далее.

MS-DOS имеет две возможности для резидентного завершения программ:

- 1) прерывание 27h (только для .COM-файлов);
- 2) функция AH=31h прерывания 21h (для .EXE- и .COM-файлов).

Небольшие различия между ними обусловлены способом, который использует MS-DOS для определения размера блока памяти, объявляемого резидентным. Исполняя прерывания 27h, MS-DOS принимает за начало блока значение в регистре CS, а длина блока в байтах задается в регистре DX. При выполнении MS-DOS-функции 31h значение в DX задает длину блока в параграфах, а за начало блока принимается идентификатор активной программы (PID) — параграф, по которому в памяти располагается префикс сегмента исполняемой в данный момент программы. В регистре AL задается код возврата, который передается программе-предку при завершении TSR.

Действия MS-DOS в случае резидентного завершения подобны тем, которые выполняются при изменении размера блока (подробнее об этом см. в 6.1 второй книги комплекса). Блок, с которого начинается PSP, усекается до запрошенного размера, но остается занятым после завершения программы. Раз так, то хранимые там данные и код программы не будут переопределяться при загрузке новых программ и (или) создании новых блоков памяти. Ведь эта память с точки зрения MS-DOS остается “занятой”. Следовательно, TSR может быть запущена в любой момент после завершения без повторной загрузки в память.

Сама TSR-программа состоит из двух функционально различных секций: резидентной и инициализирующей. *Инициализирующая часть* выполняется только однажды — при запуске программы на исполнение. *Резидентная часть TSR* чаще всего состоит из двух секций: составленных с соблюдением специальных правил одного или нескольких обработчиков прерывания, или ISR (Interrupt Service Routine), и обычных Си-функций, вызываемых из ISR. Обработчики получают управление при возникнове-

нии каких-либо внешних условий, генерирующих аппаратное прерывание, или при выполнении текущей программой инструкции INT (как говорят, при возникновении программного прерывания). Точки входа в ISR записываются в таблицу векторов прерывания (подробнее о ней см. в 1.3 второй книги комплекса). В простейшем случае вся резидентная часть TSR представляет собой единственный обработчик прерывания. Наибольший интерес при проектировании TSR представляют ISR, основы дизайна которых подробно рассматриваются в следующих параграфах.

2.1.2. Требования к программе-обработчику прерывания. Модификатор типа функции interrupt

Функция, построенная для использования в качестве ISR, должна быть абсолютно “незаметной” для тех программ, которые она прерывает или, как часто говорят, на фоне которых исполняется. Это возможно только в том случае, если состояние системных и аппаратных средств компьютера перед активизацией ISR и после ее завершения будет одним и тем же. Кроме того, для активизации ISR процессор выполняет инструкцию INT, что связано с сохранением в стеке не только CS и IP точки возврата, но и слова флагов. Поэтому ISR должна удовлетворять следующим обязательным требованиям:

- 1) ISR должна начинаться секцией кода, сохраняющей регистры процессора, изменяемые в процессе работы;
- 2) ISR должна завершаться секцией восстановления всех изменяемых в ходе своей работы регистров;
- 3) возврат из ISR должен выполняться не обычной машинной командой RET, а специальной инструкцией IRET.

Соблюдение данных требований обычными средствами Turbo C (псевдопеременные, библиотечные функции и т.п.) невозможно. Однако Turbo C предоставляет программисту исключительно удобную возможность для этого. Если обычная Си-функция объявляется с модификатором interrupt, компилятор автоматически добавляет секции сохранения всех регистров в начале кода функции,

их восстановления в конце этого кода и вместо RET подставляет IRET, обеспечивая правильный возврат из прерывания. Например, если планируется использовать функцию `my_isr( )` в качестве обработчика прерываний, ее следует описать так:

```
void interrupt my_isr(unsigned bp, unsigned di,  
    unsigned si, unsigned ds, unsigned es, unsigned dx,  
    unsigned cx, unsigned bx, unsigned ax, unsigned ip,  
    unsigned cs, unsigned flags)  
{  
    ...  
}
```

Параметры в круглых скобках задают значения соответствующих регистров, передаваемых обработчику прерывания `my_isr()`. Если регистры для передачи параметров не используются, специфицировать их не нужно. Поэтому возможно следующее описание `my_isr( )`:

```
void interrupt my_isr()  
{  
    ...  
}
```

Все функции типа `interrupt` считаются `far`-функциями, но вызываются иначе. Если при вызове обычной `far`-функции компилятор использует машинную команду `CALL FAR`, а функция завершается командой `RET FAR`, то для `interrupt`-функций использование только `CALL FAR` приведет к “повисанию”. Ведь такая функция завершается командой `IRET`, по которой из стека извлекаются значения для `CS` и `IP` точки возврата, а также слово флагов. Но из стека нельзя извлекать то, чего там не было и нет! Вот почему компилятор при обращении к `interrupt`-функции перед `CALL FAR` помещает инструкцию `PUSHF`, записывая в стек слово флагов. Это сильно облегчает построение цепочек обработчиков прерывания (см. п. 2.1.4).

Помимо сохранения регистров, при входе в `interrupt`-функцию компилятор генерирует код для установки ре-

гистра DS равным сегменту DGROUP (см. вторую книгу комплекса, 6.2). Практически это означает, что ISR имеет доступ к статическим и внешним данным программы. Использование автоматических переменных, хранимых в стеке, для функций типа interrupt возможно в случае, когда приняты дополнительные меры по переключению стека (см. п.2.3.1). Компилятор не генерирует автоматически код по переключению стека на те значения, которые соответствуют interrupt-функции.

Кроме рассмотренных требований, грамотно построенная ISR должна соблюдать другие условия, вытекающие из одного очевидного: завершив работу, ISR чисто “прибирает за собой”. Иначе прерванная программа, вновь начав свое выполнение, столкнется с неожиданными изменениями. Последствия этого при работе под управлением MS-DOS непредсказуемы. Наиболее вероятный результат — необходимость перезагрузки компьютера с выключением питания. (Обсуждение дизайна ISR будет продолжено в 2.3.)

2.1.3. Передача и возврат значений в обработчиках прерывания

В ряде случаев возникает необходимость в передаче параметров interrupt-функциям и возвращении значений. Так как основное назначение таких функций — это их использование в качестве ISR, естественно, что обмен значениями следует выполнять через внутренние регистры процессора.

Компилятор добавляет в код interrupt-функции секцию записи в стек всех внутренних регистров процессора, за исключением SS и SP. Так как interrupt-функция вызывается как обработчик прерывания (либо явно инструкцией INT, либо ее аналогом PUSHF, CALL DWORD PTR), в стеке будут сохранены флаги и адрес точки возврата. В результате после исполнения начальной (недоступной программисту) секции interrupt-функции стек будет в состоянии, которое иллюстрирует рис. 2.1.



Рис. 2.1. Состояние стека после вызова interrupt-функции

Достаточно просто решается задача передачи в interrupt-функцию параметров в регистрах. Перед вызовом функции нужно записать передаваемые значения в регистры и вызвать функцию. Для этого удобно использовать псевдопеременные `_BP`, `_DI`, `_SI` и т.п. Если точка входа в вызываемую interrupt-функцию записана в таблице векторов прерывания, передачу управления функции выполняет `geninterrupt()`. Другая возможность — это использование библиотечной функции `int86()` и подобных ей. В последнем случае решается сразу и задача установки нужных значений в регистры процессора. Но в ряде случаев interrupt-функция вызывается не через таблицу векторов прерывания, а косвенным вызовом (`CALL DWORD PTR`). Это не позволяет передавать параметры в interrupt-функцию через регистр `DS`: текущее значение `DS` используется для задания в сегменте данных адреса точки входа в функцию.

Возврат значений в регистрах из interrupt-функции возможен, но требует некоторых дополнительных усилий. Установка возвращаемого значения регистра в пределах interrupt-функции, например, через псевдопеременные `_AX`,

`_BX`, `_CX` и тому подобные не сохраняются при завершении функции. Компилятор, как отмечено в п.2.1.2, встретив модификатор типа функции `interrupt`, вставляет в начало кода функции секцию сохранения всех регистров, а в конец — секцию восстановления запомненных значений из стека. Поэтому все изменения значений регистров внутри `interrupt`-функции не будут сказываться на значениях регистров после ее завершения. Единственная возможность для возврата значений через регистры — это запись возвращаемых значений в стек. В результате при восстановлении регистров будет устанавливаться нужное значение, а не то, которое было в регистре при входе в `interrupt`-функцию. Turbo C для облегчения доступа к копиям регистров в стеке использует формальные параметры. Первый параметр списка — это ссылка на копию в стеке регистра `BP`, второй — регистра `DI`, третий — регистра `SI` и т.д. Именно в этом и заключается смысл “говорящих” формальных параметров, перечисленных при описании функции. Если `interrupt`-функция передает в регистрах несколько значений, нужно описать все предыдущие формальные параметры. Например, если планируется передача параметров в регистре `ES`, то список будет состоять из пяти элементов, где первые четыре параметра не используются.

В качестве примера передачи параметров в `interrupt`-функцию и возврата значений из нее через регистры далее приводится тестовая программа, состоящая из `main()`-функции и `interrupt`-функции `my_isr()`. В функцию `my_isr()` передаются константы в регистрах процессора `AX`, `ES` и `BX`. Функция `my_isr()` возвращает установленными в единицу флаги нуля `ZF` (6-й в слове флагов) и переноса `CF` (0-й в слове флагов), а также устанавливает в регистр `AX` значение `F0FFh`, если при вызове `my_isr()` `AX = F000h`, `ES = BX = FFFFh` и `my_isr()` не изменяет значения регистра `AX` и флагов процессора в противном случае. Для большей “убедительности” `my_isr()` “цепляется” за неиспользуемый вектор прерывания `F1h` и вызывается как обработчик прерывания функцией `intr()`.


```

/* L2_1.C */
#include <stdio.h>
#include <dos.h>
#include <conio.h>
void interrupt my_isr(unsigned bp,      unsigned di,
                      unsigned si,      unsigned ds,
                      unsigned in_es, unsigned dx,
                      unsigned cx,      unsigned in_bx,
                      unsigned in_ax, unsigned ip,
                      unsigned cs,      unsigned out_flags)
{
    if(in_ax == 0xF000 && in_es == 0xFFFF &&
        in_bx == 0xFFFF)
    {
        /* установка флагов ZF и CF */          out_flags
|= 0x0041;
        in_ax = 0xF0FF; /* возврат AX = F0FF      */
    }
}
void main(void)
{
    struct REGPACK r;
    int ch, cs, ip;
    void interrupt (* old_f1)(void); /* здесь сохраняется
        "старый" вектор прерывания f1h */
    old_f1 = getvect(0xf1);
    setvect(0xf1, my_isr); /* за прерывание f1h
        "цепляется" my_isr() */
    do
    {
        printf("AX (16c/c) ="); scanf("%x", &r.r_ax);
        fflush(stdin);
        printf("BX (16c/c) ="); scanf("%x", &r.r_bx);
        fflush(stdin);
        printf("ES (16c/c) ="); scanf("%x", &r.r_es);
        fflush(stdin);
        /* активизация my_isr через прерывание */
        intr(0xf1, &r);
        printf("AX = %#x, ZF = %1d, CF = %1d\n",
            r.r_ax, (r.r_flags & 0x40) >> 6,
            r.r_flags & 0x01);
    }
}

```



```

    puts("Завершаете? (Y/N)");
}
while(!((ch = getch()) == 'y' || ch == 'Y'));
setvect(0xf1, old_f1); /*восстанавливается вектор f1h*/
}

```

Возможность установки нужных значений в регистрах на выходе из ISR позволяет использовать interrupt-функции в качестве расширений BIOSa и MS-DOS, предотвратить повторную установку TSR и др.

2.1.4. Общий дизайн резидентной программы. Каскадное включение обработчика прерывания

TSR-программа состоит из двух частей: инициализирующей и резидентной. Инициализирующая часть TSR-программы выполняется только однажды, при инсталлировании резидентной части. Как отмечалось ранее, резидентная часть TSR состоит из двух секций: одного либо нескольких обработчиков прерывания, или ISR, и одной либо нескольких обычных Си-функций, вызываемых из ISR. *Обработчик прерывания (ISR)* — это функция, получающая управление при выполнении аппаратных или программных прерываний. В самом простом случае резидентная часть TSR представляет собой единственный обработчик прерывания. В большинстве же случаев в состав TSR входит не один, а несколько ISR, работающих параллельно и взаимодействующих друг с другом через внешние переменные, выполняющие роль бинарных семафоров. Такой семафор принимает только два значения: YES (1) и NO (0).

Инициализирующая часть выполняет обязательные и дополнительные (факультативные) действия. К числу обязательных относятся:

- 1) установление связи резидентного обработчика прерывания с тем или иным прерыванием системы (перехват прерывания);
- 2) резидентное завершение TSR.

Факультативные действия призваны повысить “профессионализм” TSR и включают:

1) предотвращение повторной установки резидентной части TSR;

2) освобождение памяти, занятой TSR;

3) перехват других прерываний системы для повышения надежности работы TSR: прерывания по нажатию клавиш Ctrl-Break, прерывания критической ошибки MS-DOS, обращения к диску, от таймера и др. Эти дополнительные действия обсуждаются как в 2.2, так и в 2.3, поскольку они связаны с изменениями в обработчике прерывания.

В дальнейшем будут рассматриваться две основные схемы включения обработчиков прерывания, входящих в состав резидентной части TSR:

1) каскадное включение;

2) переопределение “старого” обработчика.

Каскадное включение — это такая установка в систему “нового” обработчика прерывания, при которой последний получает управление в случае возникновения аппаратного или программного прерывания, а затем вызывает “старый” обработчик прерывания (рис. 2.2). “Старый” обработчик — это ISR, которая получала управление при генерации данного прерывания до запуска TSR. “Новый” обработчик — это ISR, входящая в состав резидентной программы. Он может выполнять дополнительные действия либо до вызова “старого” обработчика, либо после возврата управления из “старого” обработчика.

На рис. 2.2 показано каскадное включение “новой” ISR `my_isr()`, получающей управление при нажатии (отпускании) любой клавиши клавиатуры и выполняющей свои функции после “старого” обработчика. Считается, что исходный (“старый”) обработчик прерывания 9 — это ISR BIOS (подробнее об обработке прерываний от клавиатуры см. во второй книге комплекса). На практике для некоторых прерываний каскад включает не две, а много больше ISR. Ведь вместо ISR BIOS прерывания 9 вполне может быть ISR, вызывающая свой “старый” обработчик, который, в свою очередь, вызывает еще одну ISR. Именно для прерывания 9 обычно выстраиваются такие длин-

ные каскады. Функция `my_isr()` может быть построена так, что “новые” действия будут выполняться перед “старыми” функциями. Для этого вызов “старого” обработчика выполняется в самом конце “нового”.

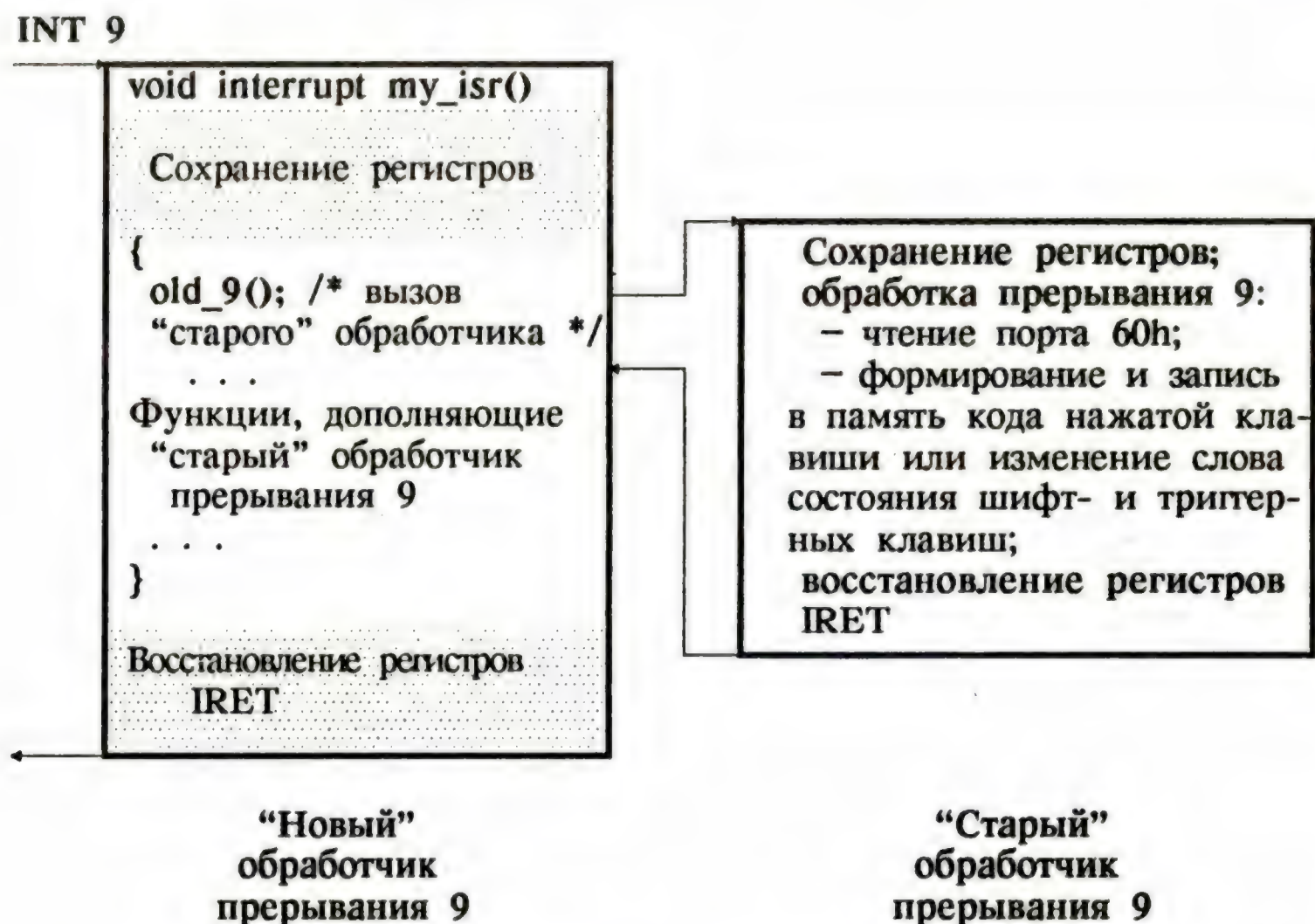


Рис. 2.2. Каскадное подключение обработчика прерывания 9 (“новые” функции выполняются после “старых”)

Переопределение — это способ подключения ISR, при котором “новый” обработчик не вызывает из своих пределов “старый”, т.е. полностью подменяет “старую” ISR. (Подробнее этот вопрос обсуждается в 2.2.)

Материал данной главы иллюстрируется сквозным примером TSR-программы, сохраняющей в файле на диске копию произвольного окна экрана в текстовом режиме. Резидентная часть TSR активизируется нажатием клавиши `PrnScr` (ПечЭкр). Имя файла для сохранения копии окна экрана задается в диалоге. Перечислим клавиши, на которые будет реагировать резидентная часть TSR:

`PrnScr` (ПечЭкр) — на экране появляется прямоугольное окно, отмечаемое контрастным цветом. Размеры окна

могут изменяться с клавиатуры. Координаты являются статическими, т.е. при следующей активизации TSR окно имеет координаты, заданные в предыдущем сеансе работы с ней;

клавиши со стрелками (Left, Right, Up, Down) — управляют координатами левого верхнего окна экрана;

клавиши Ctrl-Left, Ctrl-Right, CtrlPgUp, CtrlPgDn — управляют координатами правого нижнего угла экрана;

S (Save) — на экране появляется окно диалога, в котором задается имя файла назначения. Окно позволяет редактировать имя файла. Строка имени является статической, сохраняя значение, заданное в предыдущем сеансе работы с программой;

Esc (Ключ) — TSR возвращает управление в прерванную программу, не выполняя никаких действий;

Q (Quit) — “снятие” резидентной части TSR с освобождением занимаемой памяти.

По мере рассмотрения материала TSR будет становиться все “умнее” и надежнее. Приводимые далее примеры Си-функций рассчитаны на многофайловую компиляцию с включением в файл проекта инициализирующей функции main(), собственной ISR и всех необходимых функций — резидентного завершения, вывода строк, рамок и т.д. Ориентация на многофайловую компиляцию заставляет повторять в файлах включение файлов и явно описывать многие объекты как внешние.

Разумная технология разработки TSR на Turbo C базируется на следующем подходе: резидентные Си-функции разрабатываются как обычные с использованием всех доступных в Turbo C средств. Однако следует учитывать, что эти функции могут вызываться на фоне любых программ при самом неблагоприятном сочетании условий. Всю дополнительную работу по определению моментов безопасного вызова резидентных функций следует выполнять в пределах ISR. В результате ISR представляют собой небольшие interrupt-функции, обеспечивающие условия безопасного вызова обычных Си-функций.

2.2. Функции, выполняемые инициализирующей частью TSR-программы

2.2.1. Перехват прерывания

Перехват прерывания — это запись адреса точки входа “нового” обработчика прерывания в таблицу векторов прерывания. После того, как это действие выполнено, каждое прерывание данного номера будет активизировать “новую” ISR. Использование “старого” вектора зависит от опций поведения инсталлятора. Как правило, “старый” вектор записывается в сегменте данных TSR и используется для восстановления таблицы векторов прерывания в исходное состояние при удалении TSR из памяти и(или) при каскадном включении “нового” обработчика как адрес для вызова “старого” обработчика.

Существует несколько возможностей для записи в таблицу векторов прерывания “нового” обработчика TSR-программы:

- 1) непосредственный доступ к векторам прерывания с использованием механизма far-указателей;
- 2) применение специальных библиотечных функций Turbo C;
- 3) использование функций MS-DOS.

Непосредственный доступ к таблице векторов прерывания — наиболее быстрый способ. Однако манипуляции с векторами требуют особой осторожности. Так как аппаратное прерывание может возникнуть в любой момент при выполнении программы, вероятно ситуация неполного изменения вектора прерывания, что, конечно же, недопустимо. Для того чтобы исключить такую возможность при манипуляции с таблицей векторов, следует установить в 0 флаг IF, блокируя аппаратные прерывания, а после завершения работы с таблицей — вновь разрешить аппаратные прерывания, установив IF в 1. Для управления флагом IF можно использовать функции библиотеки Turbo C `enable()` и `disable()` соответственно для разре-

шения и блокирования аппаратных прерываний. Например, так мог бы выглядеть простейший инсталлятор TSR, который связывает interrupt-функцию ISR_int5() с вектором прерывания 5, выполняя переопределение “старого” обработчика прерывания 5. В скобках дана нумерация точек пошагового комментария выполняемых программой действий (см. далее).

```
/* L2_2.C
```

```
Инсталлятор резидентной программы, использующей  
прерывание 5. “Старый” обработчик прерывания вызывается  
перед завершением “нового”. */
```

```
#include <dos.h>
```

```
(1) struct address {void interrupt (* p) (void);};
```

```
(8) void interrupt (* old_int5)(void);
```

```
void interrupt ISR_int5(void); /* собственная ISR пре-  
рывания 5; простейший вариант – см. L2_3.C */
```

```
void tsr(char status, unsigned size); /* прототип  
функции резидентного завершения – L2_6.C */
```

```
int main(void)
```

```
{
```

```
(2) struct address far * int5 =
```

```
(struct address far * )(5 << 2);
```

```
(3) old_int5 = int5 -> p;
```

```
(4) disable();
```

```
(5) int5 -> p = ISR_int5;
```

```
(6) enable();
```

```
(7) tsr(0, 2000);
```

```
}
```

Простейшая ISR для рассматриваемого примера программы инсталляции помещена в файл L2_3.C. В приведенном варианте — это просто “заглушка”, вызывающая функцию TSR_activate(). Никакие специальные действия по определению безопасного момента активизации резидентной функции не выполняются, что делает TSR потенциальной угрозой системе. В дальнейшем будут внесены все необходимые усовершенствования для того, чтобы свести к минимуму эту угрозу.


```

/* L2_3.C
Простейший вариант ISR прерывания 5. */
#include <dos.h>
extern void interrupt (*old_int5)(void);
void TSR_activate(void); /* прототип вызываемой
                           обработчиком резидентной функции */
void interrupt ISR_int5(void)
{
    TSR_activate(); /* активизация резидентной функции */
    old_int5();      /* вызов "старого" обработчика
                     прерывания 5*/
}

```

TSR_activate() просто обозначает факт своего вызова: печатает, начиная с центра экрана, слово “Центр”. Она рассчитана на корректную работу во всех текстовых режимах EGA- и VGA-адаптеров: 25x40, 25x80, 28x50, 43x80, 50x80. Начинается обработчик с определения текущих параметров видеосистемы и записи их во внешние переменные, используемые функцией печати строки hor_prn(). Специфика TSR такова, что для вывода информации не следует использовать функции файла <stdio.h> или <conio.h>, если активизирующая ISR не анализирует состояние MS-DOS (см. п. 2.3.3). Поэтому вывод строки выполняет hor_prn() (в 2.4 второй книги комплекса приведен вариант функции L9_9.C, выводящей информацию непосредственно в видеопамять).

```

/*
L2_4.C
Простейший вариант TSR-функции, выводящей по центру
экрана слово “Центр”. Рассчитана на EGA- и VGA-
адаптеры. Для CGA внешняя переменная max_str жестко
приравнивается 24. Для монохроматического адаптера
vid_memory должна устанавливаться равной 0xb000.
*/
#include <dos.h>
#include "screen.h"
unsigned vid_memory = 0xb800, start_adr,
        max_str, max_stolb;

```



```

void TSR_activate(void)
{
    /*
        При каждом входе в функцию определяем текущие
        значения параметров видеосистемы, которые служат
        внешними переменными функции hor_prn(). */
    max_str   = * (char far *)    MK_FP(0x40, 0x84);
    max_stolb = * (char far *)    MK_FP(0x40, 0x4a);
    start_adr = * (unsigned far *) MK_FP(0x40, 0x4e);
    /*
        Вывод тестовой строки непосредственно в видео-
        буфер. Используется L9_9.C из второй книги.
    */
    hor_prn(max_str/2, max_stolb/2, "Центр", 0x1e);
    return;
}

```

Для получения загрузочного модуля TSR-программы следует создать файл проекта, который включает файлы L2_2.C, L2_3.C, L2_4.C, L2_6.C и файл L9_9.C из второй книги комплекса, содержащий текст функции hor_prn(). После завершения компиляции и компоновки будет получена программа, при запуске которой на исполнение в памяти компьютера останется резидентная порция, активируемая при нажатии клавиши PrnScr. Компиляция резидентных программ требует тщательной настройки опций. В частности, должны выключаться опции контроля переполнения стека и использования регистровых переменных. Для рассматриваемого примера (точнее, для функции hor_prn()) критичной будет и опция, определяющая символьные переменные как беззнаковые (Unsigned Chars). При разработке TSR-программ рекомендуется использовать модель памяти SMALL, так как резидентные программы не должны быть слишком большими (более 64K байт).

Кратко поясним смысл отдельных предложений функции main() программы L2_2.C, выполняющей установку резидентной части TSR:

(1) — описывается внешний шаблон address для структуры, состоящей из единственного поля по имени p. Поле содержит

указатель на функцию типа `void interrupt name(void)`. Отметим еще раз, что `interrupt`-функции считаются компилятором `far`-функциями и поэтому `p` — это `far`-указатель, который занимает 4 байта. Фактически — это вектор прерывания. Размещение вектора в памяти компьютера в точности соответствует структуре `far`-указателя: младшие два байта занимает смещение, старшие два байта — сегмент. Для упрощения доступа к таблице векторов далее будут использоваться указатели на структурные переменные по шаблону `address`;

(2) — описывается `far`-указатель `int5` на структурную переменную по шаблону `address`, и этот указатель инициализируется значением 20 (точнее, значением 00000020). Указанное в скобках явное приведение типа гарантирует корректность инициализации. Физический адрес 20 (0000:0020) — это адрес, по которому в памяти располагается вектор прерывания 5 (номер прерывания, умноженный на 4, так как каждый вектор имеет длину 4 байта);

(3) — “старый” вектор прерывания 5 записывается во внешнюю переменную `old_int5`. Она описана как указатель на функцию типа `void interrupt` (см. (8)). В дальнейшем это значение используется “новым” обработчиком прерывания для вызова с возвратом “старого” обработчика, т.е. для образования каскада (или цепочки) обработчиков прерывания 5;

(4) — маскируются аппаратные прерывания, так как все действия с таблицей векторов прерывания следует выполнять “атомарно”, т.е. без прерываний. При написании TSR-программ всегда следует исходить из возможности возникновения самого неприятного события. В данном случае — из возможности того, что клавиша `PrnScr` будет нажата именно в тот момент, когда вектор прерывания 5 изменился только наполовину;

(5) — `cory_scr` — имя устанавливаемой резидентно программы. Так как она описана как функция типа `void interrupt`, типы операндов слева и справа от оператора присваивания совпадают;

(6) — после завершения критической секции кода необходимо вновь разрешить аппаратные прерывания. В противном случае компьютер не будет реагировать на нажатия клавиш, модифицировать счетчик реального времени дня и т.п. Время, в течение которого флаг IF остается сброшенным в 0, должно быть всегда минимально возможным;

(7) — после модификации таблицы векторов прерывания выполняется резидентное завершение всей программы. Параметр, передаваемый в функцию `tsr()`, определяет число 16-байтовых параграфов, объявляемых резидентно. Если TSR “захватывает” векторы прерываний 0, 4, 5 или 6, она должна завершаться непосредственным обращением к функции MS-DOS (подробнее об этом см. в п. 2.2.2).

Другой способ работы с таблицей векторов прерывания — использование библиотечных функций Turbo C `getvect()` и `setvect()` соответственно для получения и установки значения нужного вектора прерывания. Далее приводится спецификация этих функций.

```
#include <dos.h>
```

```
void interrupt (*getvect(int intr_num)) ()
```

Функция обращается к функции АН = 35h MS-DOS для получения адреса точки входа текущего обработчика для прерывания, номер которого задает `intr_num`. Функция возвращает far-указатель на точку входа обработчика (вектор прерывания), который считается функцией типа `void interrupt name(void)`.

```
#include <dos.h>
```

```
void setvect(int intr_num, void interrupt (*isr) ())
```

Функция обращается к функции АН = 25h MS-DOS для записи в таблицу на место вектора `intr_num` far-указателя `isr` (вектора прерывания) на точку входа в “новый” обработчик прерывания. “Новый” обработчик прерывания — это функция, описанная как `void interrupt isr(void);`.

Вот как выглядит программа-инициализатор, эквивалентная L2_2.C, но использующая для работы с таблицей векторов библиотечные функции Turbo C:

```
/* L2_5.C
```

```
Инсталлятор резидентной программы, использующей  
прерывание 5. */
```



```

#include <dos.h>
void interrupt (* old_int5)(void);
void interrupt ISR_int5(void); /* собственная ISR
                               прерывания 5; простейший вариант -
                               см. L2_3.C */
void TSR(char status, unsigned size); /* функция резид-
                                       дентного завершения - L2_6.C */

int main(void)
{
    old_int5 = getvect(5); /* сохраняем "старый" вектор 5 */
    setvect(5, ISR_int5); /* записываем новый вектор 5 */
    TSR(0, 2000);
}

```

Использование библиотечных функций `getvect()` и `setvect()` делает программу нагляднее и избавляет пользователя от необходимости заботиться о сбросе (установке) флага прерываний `IF`. Плата за это — увеличение размера загрузочного модуля программы и замедление ее работы. Например, для приведенного фрагмента с библиотечными функциями `.EXE`-файл получается бóльшим приблизительно на 50 байт. Далее во всех примерах данной главы будет использован именно такой прием работы с таблицей векторов прерывания. Дело в том, что размер инсталлятора `TSR` и его скорость не играют принципиальной роли, так как эта часть `TSR` выполняется только один раз, а при грамотной установке инсталлятор исключается из резидентной части программы.

Непосредственное обращение к функциям `MS-DOS` для работы с таблицей векторов прерывания из программы на языке `Си` не дает никаких дополнительных преимуществ и далее не рассматривается.

В приведенном примере обработчика прерывания `ISR_int5()` используется каскадное построение — “новый” обработчик выполняет “новые” функции, а затем вызывает “старый” обработчик. Такое построение позволяет дополнить “старые” обработчики, в роли которых, чаще всего, выступают стандартные `ISR BIOSa` и `MS-DOS`, “новыми” функциями, делая систему открытой для расширений.

Вызов “старого” обработчика может выполняться в самом начале “нового” обработчика. В этом случае “новые” функции выполняются после возврата управления. Как частный случай, “новый” обработчик может не образовывать каскад ISR. Однако в этом случае все “старые” обработчики, в том числе возможно входящие в BIOS и MS-DOS, отключаются или как, иногда говорят, “топятся”.

Полное переопределение ISR возможно тогда, когда “новая” ISR выполняет все необходимые системные операции, например обработку ввода с клавиатуры, либо “цепляется” за неиспользуемый BIOSом или MS-DOS вектор прерывания. Установленное компанией Microsoft распределение векторов прерывания оставляет для пользовательских TSR векторы прерывания с номерами F1h — FFh. Однако какие-либо соглашения по распределению этих векторов другими резидентными программами отсутствуют. Поэтому всегда существует вероятность “утопить” ранее установленные резидентные программы, возможно активизируемые по неиспользуемому вектору. Иногда такой прием желателен либо для экономии памяти, либо для защиты программного продукта от копирования. Например, все профессиональные игровые программы используют собственные ISR для ввода с клавиатуры (прерывание 9), от таймера (прерывание 1Ch), печати экрана (прерывание 5) и других аппаратных прерываний, повышая скорость программы в несколько раз и защищая ее от “лишних” аппаратных прерываний. Полное переопределение аппаратных прерываний уменьшает также риск несанкционированного копирования текстовой информации в различных информационных базах и “расшифровку” способа защиты программы от копирования с помощью ранее установленных резидентных программ типа рассматриваемой в данной главе `coru_scr`. “Каскадное” построение — наиболее часто применяемый на практике прием.

В рассмотренном примере “старый” обработчик вызывается с возвратом в точку вызова и при этом используется машинная инструкция `CALL DWORD PTR` с косвенной адресацией, предваряемая машинной инструкцией

“пуширования” флагов. Именно так компилируется строка кода

```
old_int5();
```

в файле L2_3.C. Существуют более изящные решения по образованию каскадов обработчиков, доступные, правда, только при использовании ассемблера. Идея этих методов состоит в том, что код обработчика прерывания, включаемого в каскад, является “недостроенным”, а его завершающее построение выполняет программа инсталляции. Например, следующий фрагмент ассемблерного кода показывает, как можно организовать каскадное включение ISR:

```
; в кодовом сегменте “новой” ISR
. . . . .
; эквивалент машинной команды INT: pushf + call far c
; “недостроенной” машинной командой CALL FAR
    pushf ; вызывается обработчик,
        ; завершаемый командой IRET
    db 9A; код машинной команды CALL FAR
        ; с прямой адресацией
old_int dd ? ; 4 байта для точки входа в “старый”
        ; обработчик
. . . . .
    iret ; возврат из “нового” обработчика
```

Программа инсталляции, прочитав вектор “старого” обработчика, записывает его в кодовый сегмент “нового” обработчика со смещения, задаваемого именем old_int. После этого 5-байтовая машинная команда CALL FAR приобретает законченный вид: первый байт (9Ah) задает код операции, а четыре следующих — адрес, по которому нужно передать управление. Трудность использования приведенного метода заключается в невозможности определить средствами Turbo C значения смещения имени old_int на этапе компиляции. Поэтому приходится использовать отладчик для определения значения смещения old_int от адреса точки входа в “новый” обработчик. Рассмотрен-

ный технический прием позволяет для вызова “старой” ISR обойтись без внешней переменной (например, `old_int5()` в листинге L2_3.C), доступ к которой неизбежно требует установки в регистр DS значения DGROUP. Кроме того, вызов с прямой адресацией `CALL FAR` выполняется намного быстрее, чем вызов с косвенной адресацией `CALL DWORD PTR`. Если “новая” ISR строится по каскадной схеме, новые функции выполняются перед “старой” ISR и не требуется возврат управления из “старой” ISR в “новую”, то наиболее рациональным будет вызов “старой” ISR машинной командой безусловного перехода `JMP FAR`. В этом случае каскад ISR выглядит так, как это показано на рис. 2.3.

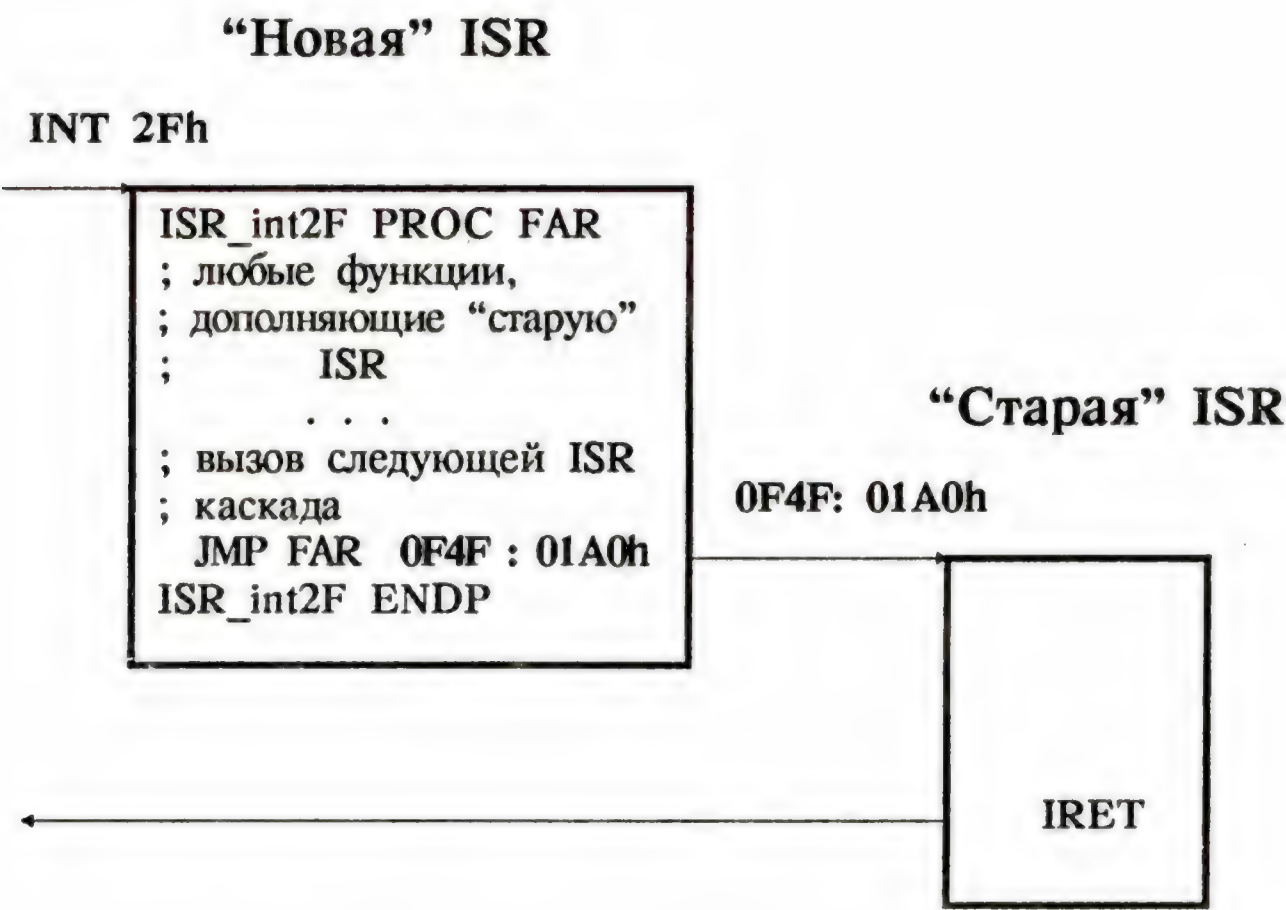


Рис. 2.3. Каскад ISR прерывания 2Fh, образованный инструкцией `JMP FAR` (листинг L2_8.ASM)

Подробный пример использования подобной техники приведен в листинге L2_11.C для “динамического” построения машинной команды `JMP FAR`, используемой в ISR прерывания 2Fh (см. листинг L2_8.ASM).

2.2.2. Резидентное завершение

Следующее после перехвата прерывания обязательное действие инициализирующей части TSR — *резидентное завершение*. Для этого можно:

- 1) использовать специальную функцию `keep()` библиотеки Turbo C;
- 2) обратиться непосредственно к функции `AH = 31h` MS-DOS.

Далее приводится спецификация функции `keep()`.

```
#include <dos.h>
```

```
void keep(unsigned char status, unsigned size)
```

Обращаясь к MS-DOS-функции `AH = 31h`, оставляет текущую программу в памяти резидентно. Резидентный блок памяти начинается с адреса `_psp:0` и занимает `size` параграфов. В префикс передается код возврата, задаваемый переменной `status`.

Выбор метода зависит от прерывания, за которое “цепляется” “новый” обработчик прерывания. Если это прерывания 0, 4, 5 или 6, функция `keep()` непригодна, так как перед обращением к функции `AH = 31h` MS-DOS резидентного завершения программы `keep()` выполняет восстановление перечисленных векторов в те значения, которые они имели перед входом в `main()`. Раз так, то все изменения этих векторов, сделанные инициализирующей частью TSR, не будут действовать после завершения программы. В этом случае приходится обращаться непосредственно к MS-DOS. Далее приводится текст функции `tsr()`, параметры которой совпадают с параметрами `keep()`, но для резидентного завершения используется обращение к операционной системе.

```
/* L2_6.C
```

```
Резидентное завершение программы. Размер резидентного  
блока – size параграфов, в MS-DOS передается код  
завершения status. Используется тогда, когда требуется  
“зацепить” TSR за прерывания 0, 4, 5 и 6. В остальных  
случаях следует использовать keep(). */
```



```
#include <dos.h>
void TSR(char status, unsigned size)
{
    _AH=0x31;
    _AL = status;
    _DX=size;
    geninterrupt(0x21);
}
```

Основная задача при резидентном завершении программ — запрос достаточного числа параграфов памяти. Если в резидентный блок не уместится весь код и данные, необходимые для работы TSR, загрузка следующих программ непременно “запортит” либо обработчик прерывания, либо его данные. Слишком большой “запас” также недопустим — резидентный блок уменьшает размер доступной для MS-DOS памяти. Самое простое решение — выбрать размер блока “на глазок”. Обычно неприятностей не бывает, если разделить размер .EXE-файла, содержащего TSR, на 8. Фактически — это запрос блока, размер которого вдвое превышает весь код TSR, включая и программу инициализации, хранить которую резидентно не нужно. К сожалению, принятый способ формирования загрузочного модуля в Turbo C не позволяет достаточно просто исключить память, занимаемую программой инсталляции.

Более точным будет метод определения размера резидентного блока памяти исходя из структуры загрузочного модуля в соответствии с моделью памяти (см. 6.2 второй книги комплекса). Для всех моделей памяти стек располагается после секции кода и статических данных. В момент обращения к функции `keep()`, если она расположена в последней строке инициализирующего кода, текущая вершина стека `SS:SP` приблизительно соответствует границе программы. Если перевести этот физический адрес в параграфы и вычесть из него сегмент, по которому начинается PSP, получится достаточно точный размер резидентной программы. Правда, для моделей памяти TINY, SMALL и MEDIUM в этот размер включается и

near-"куча", что очень сильно завышает размер резидентного блока. Но Turbo C позволяет установить предельный размер памяти, занимаемый near-"кучей". Для этого используется внешняя переменная `_heaplen`: присвоенное ей значение устанавливает предельный размер "кучи". Значение 0 соответствует "куче" максимально возможной длины. Для остальных моделей памяти, не имеющих near-"кучи", `_heaplen` не используется. И, наконец, внешняя переменная `_stklen` позволяет установить размер стека, используемый программой в процессе исполнения. С учетом сказанного будут понятны изменения, которые следовало бы сделать в программе L2_2.C:

```
/*
  L2_7.C
  Инсталлятор резидентной программы, использующей
  прерывание 5.
*/
#include <dos.h>
void interrupt (* old_int5)(void);
void interrupt ISR_int5(void); /* собственная ISR прерывания 5; простейший вариант - см. L2_3.C */
void TSR(char status, unsigned size); /* прототип функции резидентного завершения - L2_5.C */
extern unsigned _heaplen = 1024; /* задание размера "кучи" */
extern unsigned _stklen = 512; /* задание размера стека */
int main(void)
{
  unsigned size;
  old_int5 = getvect(5); /* сохраняем "старый" вектор 5 */
  setvect(5, ISR_int5); /* записываем новый вектор 5 */
  size = _SS + (_SP + 15) / 16 - _psp;
  TSR(0, size);
}
```

2.2.3. Предотвращение повторной загрузки

Как отмечено ранее, одной из специфических задач программы инициализации является определение того, оставлена ли в памяти резидентная часть TSR-программы. Если не выполнять такую проверку, при повторном

запуске, например по забывчивости или незнанию, в памяти появится второй экземпляр резидентной части TSR, затем третий, четвертый и т.д., пока не будет исчерпана вся доступная память. Решение задачи состоит в выполнении инициализирующей частью TSR проверки того, была ли установлена ранее резидентная программа и работоспособна ли она. Возможны три основных метода реализации сформулированной задачи:

1) сканирование цепочки MCB-блоков (см. п.6.5.3 второй книги комплекса) и определение того, имеется ли в памяти блок, “хозяином” которого является TSR с известным именем и(или) маршрутом расположения в файловой иерархии;

2) использование специальных сигнатур, помещенных в ISR;

3) использование “эха” в цепочке ISR.

Первый из методов позволяет обнаружить наличие в памяти резидентной части TSR, но не дает ответа на вопрос, работоспособна ли она.

Идея второго метода состоит в следующем. В ISR в заранее известном месте помещается идентифицирующий код подобно тому, как это было сделано в примере использования функции `__emit__()` (см. п.1.1.1). Смещение `sign_offs` до сигнатуры определяется просмотром машинного кода в отладчике. Точкой отсчета для смещения будет служить вектор прерывания, за который “цепляется” ISR. Процедура проверки состоит в следующем:

1) из таблицы считывается вектор прерывания, который используется ISR;

2) к смещению прибавляется `sign_offs`;

3) формируется far-указатель, сегмент которого равен сегменту вектора прерывания, а смещение вычислено на шаге 2 ;

4) проверяется, совпадают ли байты в памяти по вычисленному указателю с байтами строки сигнатуры. Если совпадение полное, это означает, что в данный момент за вектор прерывания “зацеплена” ISR и повторную установку выполнять не следует. Несовпадение свидетельствует о том, что за вектор прерывания “зацепле-

на” “старая” ISR либо “новая” ISR была “перцеплена”. В неоднозначности и кроется основной недостаток метода — он не способен определить, установлена ли уже собственная ISR, если она не первая в каскаде.

Третий метод — метод “эха” — избавлен от этого недостатка. Его идея проста. В одну из собственных ISR при разработке включается дополнительная секция — так называемый “фильтр”. Эта ISR подключается обязательно по каскадной схеме. Получив управление, “фильтр” сравнивает текущие значения внутренних регистров процессора с “эхо”-константами. Если совпадение найдено, “фильтр” изменяет на заранее известные значения некоторых регистров или флагов (так называемый “эхо”-возврат). Для проверки наличия в памяти “резидента” программа инициализации активизирует обработчик прерывания, передавая ему “эхо”-константы. “Фильтр” устанавливает в регистры и(или) флаги “эхо”-возврат и вызывает “старую” ISR каскада обработчиков. Так как все правильно построенные ISR сохраняют неизменными регистры, то “эхо”-возврат придет в инициализатор неизменным всеми обработчиками цепочки. Если ISR отсутствует в цепочке обработчиков, программа инициализации не обнаружит в регистрах “эхо”-возврат и будет полагать, что собственная ISR еще не установлена. Метод работает для собственных ISR, “перцепленных” другими резидентными программами сколь угодно далеко от начала каскада. Единственный недостаток метода — вероятность того, что значения “эхо”-констант могут совпасть со значениями регистров, означающими для других ISR выполнение тех или иных действий. Однако, выбрав “экзотическую” комбинацию для “эхо”-констант, эту вероятность можно практически свести к нулю (см., например, L2_1.C). При выборе “эхо”-возврата не следует изменять значения аппаратных регистров, за исключением АХ, так как это чревато непредсказуемыми последствиями.

Метод “эха” нашел наибольшее распространение среди профессиональных резидентных программ. Для размещения “фильтра” наиболее часто используются:

1) BIOS ISR управления аппаратурой, например прерываний 10h, 13h, 17h и др. В качестве “эхо”-константы выбирается значение, не совпадающее ни с одним из номеров функций BIOSa данного прерывания. Например, для прерывания 13h (Управление диском) любое значение AH, большее 1Bh, может надежно использоваться для этих целей. Недостаток метода состоит в значительном уменьшении системной производительности, так как каскады часто вызываемых обработчиков прерываний удлиняются;

2) ISR прерывания 2Fh, называемого *MS-DOS-мультиплексным прерыванием*. Каскад обработчиков этого прерывания является стандартным средством, рекомендуемым MS-DOS для взаимодействия с резидентными программами, в том числе входящими в состав MS-DOS: APPEND, PRINT, ASSIGN, SHARE. Значение, устанавливаемое в регистре AH, используется MS-DOS для индикации резидентной программы, которой адресуется запрос. Значение, устанавливаемое в AL, задает код выполняемой команды (табл. 2.1).

Обработчик прерывания 2Fh строится по каскадной схеме и при получении управления анализирует значения в регистре AH. Если оно равно закрепленному за пользовательской ISR значению, выполняется запрошенная команда. Для определения того, установлена ли уже резидентная порция TSR, используется команда AL=0. Правила построения “фильтров” прерывания 2Fh таковы, что, получив такую команду, “фильтр” сообщает о состоянии резидентной программы. Отсюда следует алгоритм определения наличия в памяти собственной резидентной программы:

1) инициализирующая часть TSR устанавливает AH равным номеру, выбранному для ISR (в диапазоне C0h—FFh), а AL — нулю и выдает прерывание 2Fh;

2) на выходе из прерывания 2Fh анализируется значение регистра AL: если оно не изменилось, TSR не установлена; если AL=FFh, резидентная программа уже установлена и повторная установка не требуется.

Табл. 2.1. Мультиплексное прерывание (INT 2Fh)

Значение	Описание
На входе в прерывание	
AH	Индикатор вызываемой резидентной программы
01h	Резидентная часть команды PRINT MS-DOS
02h	Резидентная часть команды ASSIGN MS-DOS
03h	Зарезервировано MS-DOS (недоступно для использования)
...	
0Fh	Зарезервировано MS-DOS (недоступно для использования)
10h	Резидентная часть команды SHARE MS-DOS
11h	Зарезервировано MS-DOS (недоступно для использования)
...	
42h	Зарезервировано MS-DOS (недоступно для использования)
43h	Драйвер расширенной памяти (XMS-драйвер, см. 7.4)
44h	Зарезервировано MS-DOS (недоступно для использования)
...	
B6h	Зарезервировано MS-DOS (недоступно для использования)
B7h	Резидентная часть команды APPEND MS-DOS
B8h	Зарезервировано MS-DOS (недоступно для использования)
...	
BFh	Зарезервировано MS-DOS (недоступно для использования)
C0h	Доступно для использования резидентной программой
...	
FFh	Доступно для использования резидентной программой
AL	Команда, передаваемая резидентной программе
00h	Состояние резидентной программы
...	
10h	Запрос адреса точки входа в функции XMS-драйвера (если AH=43h)
На выходе из прерывания	
AX	Код ошибки для резидентных программ MS-DOS, если установлен флаг переноса CF
AL	Состояние резидентной программы
00h	Программа не установлена резидентно и ее можно устанавливать
01h	Программа не установлена резидентно и ее нельзя устанавливать
FFh	Программа установлена резидентно

Преимуществом рассмотренного метода является то, что не задействуется дополнительный вектор прерывания для использования “фильтром”. Другое достоинство ме-

тогда заключается в том, что MS-DOS генерирует прерывание 2Fh при выполнении многих внутренних функций, например при смене буквы текущего накопителя, и, если резидентные программы MS-DOS не установлены, собственная резидентная программа будет получать управление.

В данной главе используется метод “эха”, а программа “фильтр” включается в каскад обработчиков прерывания 2Fh. Программе присвоен номер F0h и, следовательно, “эхо”-константе — значение в регистре AX = F000h. “Эхо”-возвратом будет значение в регистре AX, равное F0FFh. Обработчик, включаемый в каскад прерывания 2Fh, строится по особым правилам, соблюдение которых невозможно при программировании на Си. Поэтому далее приводится пример ISR, включаемой в каскад прерывания 2Fh, написанный на ассемблере:

```
; L2_8.ASM
; ISR, включаемая в каскад обработчиков прерывания 2Fh.
; Используется для предотвращения повторной установки
; резидентной программы в памяти.
```

```
_TEXT SEGMENT BYTE PUBLIC 'CODE'
DGROUP GROUP _DATA
_TEXT ENDS
PUBLIC _ISR_int2F; делает идентификатор видимым из Си
.MODEL SMALL
```

```
.CODE
_ISR_int2F PROC FAR
    cmp ax, 0F000h; вызвана ли резидентная
                    ; программа ?
    jne chain_2F  ; нет; переход на “старый”
                    ; обработчик
    xor al, 0FFh  ; установка “эхо”-возврата
    iret          ; выход
; вызов следующей ISR каскада машинной командой
; безусловного перехода JMP FAR. Сама команда “дост-
; раивается” программой инсталляции.
chain_2F:        db 0EAh ; код команды JMP FAR
```



```

_old_vector2F dd ?      ; место для адреса "старой" ISR
                  ; смещение до этой точки
                  ; равно 9 байт

_ISR_int2F      ENDP
                END

```

Занесение "старого" вектора прерывания 2Fh `_old_vector2F` в кодовый сегмент выполняет программа инсталляции, используя функцию `movdata()`. Для этого в отладчике необходимо определить смещение до метки `_old_vector2F`. При компиляции приведенного примера с использованием модели памяти `SMALL` получено значение 9.

Далее приводится функция `is_installed()`, которая реализует рассмотренный метод. Функция возвращает `YES` (1), если получен "эхо"-возврат и `NO` (0) в противном случае.

```

/*
  L2_9.C
  Определение того, присутствует ли в каскаде ISR
  прерывания 2Fh "фильтр", реагирующий на значение в
  AX=0xF000 установкой AL=0xFF.
*/
#include <dos.h>
#define YES    1
#define NO     0

int is_installed(void)
{
    union REGS r;
    /* Установка "эхо"-константы. */
    r.x.ax = 0xF000;
    int86(0x2F, &r, &r);
    /* Анализ "эхо"-возврата. */
    if(r.h.al == 0xFF)
        return YES; /* резидентная часть
                     находится в памяти */
    else
        return NO;  /* резидентная часть отсутствует */
}

```


Теперь можно написать вполне профессиональную программу инсталляции. "Правила хорошего тона" требуют выдать на печать сообщение об установке резидентной программы и дать краткую инструкцию об использовании. Если инсталлятор обнаруживает, что TSR уже установлена, выдается сообщение об этом и повторяется вывод инструкции об использовании. Для рассматриваемой программы копирования окна экрана вывод на экран инструкции об использовании будет выполнять функция `print_help()`.

```
/*
    L2_10.C
    Выводит инструкцию об использовании резидентной
    программы. */
#include <io.h>
void print_help(void)
{
    int index;
    char * icons[] = {
        "PrnScr",
        "Left, Right, Up, Down",
        "Ctrl+Left,+Right,+PgUp,+PgDn",
        "S (Save)",
        "Esc",
        "Q (Quit)",
        "", 0};
    char * text[] = {
        "- активизация программы;\n",
        "- управление левым верхним углом копируемого окна;\n",
        "- управление правым нижним углом копируемого окна;\n",
        "- запись текста окна в файл на диске;\n",
        "- выход без сохранения копии окна;\n",
        "- удаление программы из памяти\n" };
    printf("Резидентная программа копирования"\
        "текстового окна экрана.\n\n");
    for(index = 0; icons[index]; index++)
    {
        write(1, icons[index], strlen(icons[index]));
        write(1, text[index], strlen(text[index]));
    }
}
```


Далее приводятся текст “поумневшего” инсталлятора, проверяющего методом “эха” наличие в памяти резидентной порции TSR. “Фильтр” устанавливается в каскад прерывания 2Fh. Приводимый пример является доработкой листинга L2_7.C: добавлен вызов функции `is_installed()`, а также установка в каскад обработчиков прерывания 2Fh “фильтра”, выполняющего “эхо”-возрат.

```
/*
    L2_11.C
    Инсталлятор резидентной программы, использующей
    прерывание 5.
*/
#include <dos.h>
#include <stdio.h>
#define YES    1
#define NO     0
void interrupt (* old_int5)(void);
void interrupt (* old_int2F)(void);
void interrupt ISR_int5(void); /* собственная ISR прерывания 5; простейший вариант – см. L2_3.C */
void interrupt ISR_int2F(void); /* собственная ISR прерывания 2F, простейший вариант – см. L2_8.ASM */
void tsr(char status, unsigned size); /* функция резидентного завершения – см. L2_6.C */
int is_installed(void); /* проверка, есть ли в каскаде ISR прерывания 2Fh собственная ISR, – см. L2_9.C */
void print_help(void); /* вывод инструкции о TSR – см. L2_10.C */

/* задание размера “кучи” */
extern unsigned _heaplen = 1024;
/* задание размера стека */ extern unsigned _stklen = 512;

int main(int argc, char ** argv)
{
    unsigned size;
    /* Предотвращение повторной установки TSR. */
    if(is_installed() == YES)
    {
        printf("\n\аПрограмма %s уже установлена.\n\n",
```



```

        argv[0]);
    print_help();
    return 1;
}
/* Включение в каскад ISR прерывания 5 ISR_int5(). */
old_int5 = getvect(5);
setvect(5, ISR_int5);
/* Сохранение "старого" вектора 2Fh
   в сегменте данных. */
old_int2F = getvect(0x2F);
/* Перенос "старого" вектора 2Fh
   в кодовый сегмент ISR_int2F().*/
movedata(_DS,&old_int2F,_CS,FP_OFF(ISR_int2F) + 9, 4);
/*Включение в каскад ISR прерывания 2Fh ISR_int2F().*/
setvect(0x2F, ISR_int2F);
/* Резидентное завершение программы. */
size = _SS + (_SP + 15) / 16 - _psp;
print_help();
tsr(0, size);
}

```

Для получения загрузочного модуля TSR-программы следует сформировать файл проекта и выполнить много-файловую компиляцию. Файл проекта должен включать следующие файлы: L2_3.C, L2_4.C, L2_6.C, L2_8.ASM, L2_9.C, L2_10.C, L2_11.C и файл L9_9.C из второй книги комплекса, содержащий текст функции hor_prn(). После завершения компиляции и компоновки будет получена программа, при запуске которой на исполнение в памяти компьютера остается резидентная порция, активизируемая при нажатии клавиши PrnScr. Попытка повторной установки приведет к выдаче на экран сообщения "Программа ... уже установлена", сопровождаемого звуковым сигналом и повторением инструкции об использовании программы. Эта инструкция приведена "на вырост" для уже завершенного примера, а пока TSR просто обозначает свое присутствие в памяти выводом строки текста "Центр" по центру экрана во всех текстовых режимах.

2.3. Построение резидентной части TSR

2.3.1. Переключение стека

Резидентная часть TSR — это одна или несколько функций и взаимодействующих обработчиков прерываний. Далее для экономии места аббревиатура TSR используется для обозначения только резидентной части TSR. Естественно желание программиста для разработки резидентных функций и ISR использовать язык Си и все привычные средства библиотеки. Реализация такого подхода на практике возможна, но требует соблюдения при построении TSR ряда дополнительных условий, подробно рассматриваемых далее. Наиболее целесообразно, по мнению автора, использовать такое построение TSR:

1) обработчики прерываний получают управление и анализируют необходимость и возможность активизации TSR;

2) если активизация возможна, вызывается специальная функция, например `TSR_prepare_activate()`, выполняющая подготовительные действия, обусловленные неприспособленностью функций библиотеки Turbo C к резидентной работе;

3) после завершения всех подготовительных операций вызывается обычная Си-функция, написанная без соблюдения каких-либо особых условий и, возможно, использованная ранее в других программах. Назовем ее условно `TSR_body()`.

В идеале, получив управление, TSR должна сделать все для того, чтобы MS-DOS восприняла ее как обычную текущую программу. Перед своим завершением TSR должна чисто “прибрать” за собой. Далее программу, которую прерывает TSR, будем называть *фоновой*. Дополнительные действия TSR после получения управления в большой мере зависят от того, какие функции выполняются резидентно: используется ли доступ к файлам, выполняется ли вывод на экран и ввод информации с клавиатуры и т.д.

Программа при исполнении обязательно нуждается в стеке для сохранения адресов точек возврата, передачи параметров и др. Когда ISR получает управление, регистры SS и SP процессора указывают на текущую вершину стека фоновой программы. Если оставить их без изменения, TSR будет “паразитировать” на стеке фоновой программы. Это опасно, так как прерванная программа рассчитывает размер стека только для собственных нужд. Поскольку рассматриваемая в данной главе TSR составляется на языке Си, компилятор которого генерирует код с активным использованием стека, весьма вероятна ситуация перегрузки стека прерванной программы. Единственный вероятный результат этого — “повисание” компьютера при выходе из обработчика прерывания. Вот почему первым обязательным требованием к TSR_prepare_activate() является переключение стека. Приводимый далее Си-подобный код иллюстрирует действия по переключению стека. Как и прежде, предполагается многофайловая компиляция программы; устанавливающая часть TSR размещается в файле MAIN.C, а текст функции TSR_prepare_activate() — в файле TSR_ACT.C.

```
/* Файл MAIN.C */
```

```
...
```

```
unsigned TSR_ss, TSR_sp;
```

```
/* значения SS, SP для работы TSR */
```

```
unsigned TSR_stack[512];
```

```
/* используется под стек TSR */
```

```
unsigned old_ss, old_sp;
```

```
/* “старые” SS, SP при входе в TSR */
```

```
...
```

```
main()
```

```
{
```

```
...
```

```
TSR_ss = FP_SEG((unsigned far *)TSR_stack);
```

```
TSR_sp = FP_OFF(TSR_stack) + 1024; /* стек растёт вверх */
```

```
...
```

```
}
```



```

/* Файл TSR_ACT.C */
...
extern unsigned TSR_ss, TSR_sp, old_ss, old_sp;
...
void TSR_prepare_activate(void)
{
    /*
    Переключение стека. Это первое, что следует сделать.
    */
    old_ss = _SS;
    old_sp = _SP;
    disable(); /* установку новых SS,SP не прерывать */
    _SS = TSR_ss;
    _SP = TSR_sp;
    enable();
    ...
    TSR_body(); /* обычная Си-функция */
    ...
    /*
    Обратное переключение стека.
    */
    disable(); /* установку новых SS,SP не прерывать */
    _SS = old_ss;
    _SP = old_sp;
    enable();
}

```

Установку новых значений SS и SP обязательно выполняют с маскированными внешними прерываниями. В противном случае не исключена вероятность того, что TSR будет прервана другой резидентной программой, которая, возможно, пользуется чужим стеком. Но она застанет стек на “полпути” к переключению. Место под стек может быть зарезервировано в сегменте данных в виде статического массива, а может использоваться тот стек, который входит в загрузочный модуль Си-программы. Так как при вычислении размера резидентного блока мы исключили стек Turbo C, в примере под стек TSR-программы используется массив в 512 слов (unsigned ISR_stack[512]).

Если возникает повторное вхождение в TSR до ее завершения (т.е. если TSR может прерываться запуском самой себя), переключение стека на один и тот же массив неизбежно приведет к ошибке. При повторном вхождении в стек будут “затерты” данные, относящиеся к первому, еще не завершённому вхождению. Для того чтобы избежать такой ошибки, используется блокирование повторного вхождения в TSR. Самое простое решение при организации блокирования — использование *бинарного семафора* (специальной внешней переменной, принимающей два значения: YES и NO).

Если interrupt-функция возвращает значения в регистрах, она описывается как имеющая формальные параметры (см. п. 2.1.3). В этом случае все операции по доступу к ним должны выполняться на непереключённом (“старом”) стеке. Это следует из того, что формальные параметры являются ссылками на копии регистров, помещённые в текущий стек ещё до того, как может быть выполнено переключение стека в interrupt-функции.

2.3.2. Переключение PSP

Идентификатором активной программы для MS-DOS служит PID — сегмент, по которому в памяти располагается PSP программы. В PSP сосредоточена вся информация, необходимая MS-DOS для управления памятью и программами. Подробно структура PSP и использование его полей рассмотрены в 5.1 второй книги комплекса. Для TSR, использующей доступ к файлам, особенно важным является таблица открытых файлов PSP (см. 2.7 второй книги комплекса). В частности, для рассматриваемого в данной главе примера TSR требуется выполнять запись текстового окна в файл, а следовательно, файл должен быть открыт. Его префикс будет помещён в PSP. Даже если открытие файла выполнит программа инициализации и оставит его открытым, а префикс поместит во внешнюю переменную, это всё равно не позволит ISR использовать его! Ведь в момент, когда TSR получает управление, активной является уже другая программа, со

своим PSP и своей таблицей открытых файлов. Использование того же префикса вызовет доступ к совсем другому файлу и может его просто разрушить. Альтернативное решение — открыть нужный файл на входе в TSR, затем выполнить перенос данных, после чего вновь закрыть файл. Однако может случиться так, что у фоновой программы, на PSP которой TSR “паразитирует”, уже исчерпан лимит на префиксы. Значение лимита по умолчанию равно 20, но может быть увеличено обращением к MS-DOS (см. 2.7 второй книги комплекса). Однако TSR не имеет права делать это без ведома прерванной программы. Таким образом, надежность работы TSR снижается.

Похожая проблема возникает и при выгрузке резидентной программы из памяти, перехвате прерываний от нажатия клавиш Ctrl-Break и критической ошибки MS-DOS (эти проблемы подробно обсуждаются далее). Радикальное решение — выполнение переключения PSP. MS-DOS поддерживает специальные функции, позволяющие выполнить переключение: функция AH = 62h (GetPID) возвращает в регистре BX текущее значение PID; функция AH = 50h (SetPID) устанавливает в качестве текущего новый PID, задаваемый в регистре BX. Функция AH = 50h не документирована, но в версиях MS-DOS 3.0 и старше работает надежно. Отметим, что функции GetPID и SetPID можно выполнять только тогда, когда фоновая программа — не MS-DOS. Способ определения этого обстоятельства рассматривается в п. 2.3.3.

Приводимый далее Си-подобный код иллюстрирует действия, выполняемые по переключению PSP.

```
/* Файл MAIN.C */  
...  
unsigned old_PSP; /* хранит PID фоновой программы */  
...  
main()  
{  
    ...  
}
```



```

/* Файл TSR_ACT.C */
...
extern unsigned _psp, old_PSP;
void TSR_prepare_activate(void)
{
    ...
    /*
    Переключение текущего PSP на
    PSP резидентной программы.
    */
    _AH = 0x62;
    geninterrupt(0x21);
    old_PSP = _BX;
    _AH = 0x50;
    _BX = _psp;
    geninterrupt(0x21);
    ...
    TSR_body();          /* обычная Си-функция */
    ...
    /*
    Обратное переключение PSP
    на PSP фоновой программы.
    */
    _AH = 0x50;
    _BX = old_PSP;
    geninterrupt(0x21);
    ...
}

```

2.3.3. Предотвращение повторного вхождения в MS-DOS

В п.2.1.2 обсуждались обязательные требования, которым должна удовлетворять программа-обработчик прерывания. В п. 2.3.1 они были дополнены требованием блокировать повторное вхождение в собственную TSR. Проще всего блокировать повторное вхождение, используя внешнюю *переменную-семафор* (busy), принимающую всего два значения, например YES (0) и NO (1). Перед возможным вхождением в TSR проверяется флаг busy, и, если он равен, например, YES, вхождение в резидентную

программу не выполняется. В противном случае флаг `busy` устанавливается в `YES` и вызывается `TSR`.

Следующее обязательное требование к надежно работающей `TSR` — предотвращение повторного вхождения в `MS-DOS`. Однопрограммная `MS-DOS` не является реентерабельной. *Реентерабельная программа* — это программа, которая разрешает, в силу особенностей своего построения, начинать ее выполнение несколько раз, не дожидаясь завершения выполнения (выхода) программы, начатого ранее. Реентерабельная программа не изменяет ни одной константы или переменной, которые могут повлиять на повторное выполнение программы. Большинство программ, образующих в совокупности ядро `MS-DOS`, не являются реентерабельными (по крайней мере, для версий `MS-DOS` младше 5.0). В этой связи не являются реентерабельными и программы, обращающиеся к функциям `MS-DOS` непосредственно, либо через функции библиотеки `Turbo C`. Для `TSR`, написанной на языке `Си` и использующей библиотечные функции, всегда существует вероятность повторного вхождения в `MS-DOS`, так как `TSR` может получить управление в любой момент, в том числе и тогда, когда `MS-DOS` выполняет нереентерабельную секцию своего кода. Отсюда следует требование к `ISR` активизировать `TSR` только тогда, когда `MS-DOS` позволяет повторное вхождение. Для этого существуют две основные возможности:

- 1) использование прерывания `28h`;

- 2) использование флага повторного вхождения в `MS-DOS`.

Механизм прерывания `28h`, который часто называют “секретом прерывания `28h`”, таков. Как известно, `MS-DOS` имеет утилиты “фонового” или псевдопараллельного исполнения. Например, `PRINT.COM` позволяет печатать файлы и исполнять другую программу. Утилиты параллельного исполнения перехватывают “прерывание” `28h`. Для того чтобы параллельная программа периодически получала управление, `MS-DOS` в те моменты, когда она выполняет цикл ожидания ввода с клавиатуры и перед исполнением функций с номерами от `0h` до `0Ch`, выдает прерывание `28h`.

Если включить собственную ISR в каскад обработчиков прерывания 28h, она будет периодически получать управление и анализировать необходимость активизации TSR, например, по состоянию флага, которым управляет еще одна ISR. Ценность такого подхода к построению TSR как раз в том, что если вызывается прерывание 28h, гарантируется, что MS-DOS находится в безопасном для прерывания (а следовательно, и для возможного повторного вхождения) состоянии. Можно смело активизировать TSR, если в этом возникает необходимость. Единственное ограничение, накладываемое на TSR, получающую управление из прерывания 28h, состоит в запрете на использование функций MS-DOS от 00h до 0Ch. Это вызывает повторное вхождение в ISR прерывания 28h, и, если код ISR нереентерабельный, система “повисает”. Однако бинарный семафор в самом начале ISR (см. далее) легко решает эту проблему.

Другой способ предотвращения повторного вхождения в MS-DOS построен на использовании флага повторной входимости MS-DOS. Адрес этого флага возвращает в ES:BX недокументированная функция AH = 34h MS-DOS. Если MS-DOS находится в нереентерабельной секции своего кода, флаг повторной входимости равен 1. Если же MS-DOS исполняет реентерабельную секцию своего кода, флаг равен 0 и резидентная программа может безопасно активизироваться и использовать в своей работе все функции операционной системы. Далее приводится фрагмент кода, поясняющего определение и использование флага повторной входимости в MS-DOS.

```
/* Файл MAIN.C */
...
char far * inside_DOS_ptr; /* Адрес флага повторной
                             входимости MS-DOS */
...
main()
{
    ...
    _AH = 0x34;
    geninterrupt(0x21);
}
```



```

    inside_DOS_ptr = MK_FP(_ES, _BX);
    ...
}

...
/* В файлах ISR, выполняющих проверку флага
   повторной входимости в MS-DOS. */
if(* inside_DOS_ptr == 1)
{
    /* Действия в случае, когда запрещена
       активизация TSR. */
}
else
{
    /* Действия в случае, когда возможна активизация TSR,
       использующей в своей работе функции MS-DOS. */
}

```

Недостаток этого подхода заключается в том, что решение не активизировать TSR принимается независимо от того, какую из своих функций выполняет MS-DOS. Если флаг повторной входимости равен 1, но MS-DOS занята выполнением функций консольного ввода-вывода 01 — 0Ch, остальные функции, связанные с доступом к файлам, могут использоваться абсолютно безопасно. Поэтому данный подход можно усилить, если поставить ISR-фильтр на прерывание 21h. Его основная задача — отражать во внешних переменных-флагах текущее состояние MS-DOS. Например, работать с флагами `inside_01_0C` и `inside_0D_FF`, показывающими, выполняет ли MS-DOS в момент аппаратного прерывания функции 01 — 0Ch, связанные с консольным вводом-выводом, или какие-либо другие функции. Если выполняется консольный ввод-вывод, можно безопасно осуществлять доступ к файлам, и наоборот. Детальный анализ этого флага позволяет уменьшить задержку между наступлением момента активизации TSR и ее фактической активизацией. Если полагаться только на значение флага повторной входимости в MS-DOS (*inside_DOS_ptr), время ожидания может стать недопустимо большим. Кроме того, так как флаг повтор-

ной входимости — недокументированная возможность MS-DOS, фирма Microsoft при разработке очередной версии операционной системы может изменить способ обращения к нему или даже значение самого флага. Отметим, что составление программы-фильтра прерывания 21h на языке Си неэффективно. Ряд специфических трудностей работы с прерыванием (возможное разрушение регистров SS, SP, DS, высокая вероятность повторной входимости в код и др.) заставляют выбрать ассемблер. Флаги в этом случае следует помещать в кодовом сегменте “нового” обработчика, а это усложняет редактирование связей с другими модулями программы и делает тексты трудными для восприятия.

В случае, когда обработчик прерывания обнаруживает необходимость активизации TSR, но MS-DOS занята, возникает “отложенный” запрос активизации. Такой запрос при построении многих TSR следует обработать как можно быстрее, иначе вновь поступивший запрос “затрет” его. Универсальное решение — использовать фильтры часто вызываемых прерываний: таймера, обращения к диску и других для того, чтобы увеличить число “оказий” для активизации TSR. В рассматриваемом в 2.5 примере резидентной программы будет использоваться фильтр прерывания 1Ch. В итоге TSR строится как несколько параллельных ISR, взаимодействующих через внешние переменные: ISR прерывания 5, ISR прерывания 28h, ISR прерывания 1Ch и др. Наиболее простая и надежная схема построения TSR приведена на рис. 2.4.

Активизирующая ISR (в рассматриваемом примере это ISR прерывания 5) выполняет только установку флага, управляющего активизацией TSR. Например, пусть флаг имеет имя request. Всю остальную работу, возложенную на TSR, исполняет обработчик прерывания 28h. Он проверяет значение флага и, если request = YES, ISR анализирует возможность активизации. Препятствиями для этого могут быть:

- 1) то обстоятельство, что MS-DOS находится в критической секции своего кода;

2) ранее начатое и незавершенное исполнение функций TSR (незавершенное вхождение в TSR).



Рис.2.4. Упрощенная схема построения TSR (активизирующее прерывание 5 (печать экрана), рабочее прерывание 28h)

Если MS-DOS активизирует прерывание 28h, это свидетельствует о том, что ядро операционной системы в реентерабельном состоянии и дополнительная проверка содержимого байта памяти по адресу `inside_DOS_ptr` не требуется. Функция `TSR_prepare_activate()` выполняет все подготовительные действия и вызывает основную функцию. Ею может быть практически любая Си-функция, в том числе и выполняющая доступ к файлам на диске.

Будем считать далее, что эта функция имеет имя `TSR_body()`. (Отметим однако, что рассмотренные ранее действия по переключению стека и PSP еще не гарантируют безопасного использования любых Си-функций, а требуют других действий, описанных далее: переключения области переноса диска, перехвата прерываний критической ошибки MS-DOS и нажатия клавиш Ctrl-Break). Очевидно и требование подключать функцию `TSR_body()` через файл проекта вместе с обработчиками прерываний, программой инсталляции и функцией `TSR_prepare_activate()`. Резидентная работа функции требует “аккуратного” обращения с экраном: если параметры видеосистемы (режим, позиция курсора, активная видеостраница) будут изменяться, то требуется их восстановление перед завершением функции `TSR_body()`. Если выполняется вывод на экран, в том числе и “эхоирование” ввода с клавиатуры, предыдущее содержимое видеобуфера должно сохраняться внутри TSR, а перед завершением `TSR_body()` восстанавливаться.

По подобной схеме взаимодействия “активизирующего” и “рабочего” прерываний строятся многие популярные TSR. В качестве активизирующего могут выступать аппаратные прерывания (наиболее часто — прерывание 9 от клавиатуры; особенности построения ISR этого прерывания рассмотрены в 2.6), либо программные прерывания. В последнем случае фоновой будет программа, выполнившая инструкцию `INT`. Построение таких TSR намного проще, так как фоновая программа сама может выполнить необходимые подготовительные действия по активизации TSR, а поведение TSR не является для фоновой программы неожиданным.

Далее приводится текст ISR прерывания 28h, работающей по схеме, приведенной на рис. 2.4.

```
/*
```

```
  L2_12.C
```

```
  Фильтр прерывания 28h. Если имеется “отложенная” попытка активизации TSR, он вызывает TSR_prepare_activate().
```

```
*/
```



```

#define      NO      0
#define      YES     1
#define      OK      0

#include <dos.h>
extern void interrupt (*old_int28)(void);
extern unsigned busy, request;
int TSR_prepare_activate(void);
void interrupt ISR_int28(void)
{
    old_int28(); /* вызов "старой" ISR прерывания 28h */
    disable();   /* маскирование аппаратных прерываний */
    if(busy == NO && request == YES)
    { /* активизация TSR*/
        busy = YES;
        if(TSR_prepare_activate() == OK)
            request = NO;
        busy = NO;
    }
    return;
}

```

2.3.4. Защищенный обмен с диском. Своппинг

Если резидентной программе требуется обмен с диском, необходимо предпринять дополнительные меры безопасности. Они связаны, во-первых, с определением момента, когда безопасно могут выполняться функции файловой системы MS-DOS и опирающиеся на них библиотечные функции Turbo C, во-вторых, с созданием "привычной" для файловых функций среды.

Обращение к диску следует выполнять в тех случаях, когда флаг повторного вхождения в MS-DOS не равен 0 и не выполняется какая-либо из функций прерывания 13h. Функции этого прерывания управляют работой дисковой подсистемы на физическом уровне. К сожалению, отсутствует какое-либо системное средство определения того, активно ли в данный момент прерывание 13h. Единственная возможность — использовать фильтр прерывания 13h. Основная задача фильтра — отражать во

внешней переменной `inside_BIOS` текущее состояние: при входе в каскад ISR прерывания `13h` устанавливать флаг в состояние `YES`, на выходе — `NO`. Значение флага следует проверять в дополнение к флагу повторной входимости в `MS-DOS` и не активизировать `TSR` до завершения прерывания `13h`. Устанавливаемый фильтр должен “прозрачно пропускать” возможные значения, возвращаемые “старым” обработчиком `13h`. Возвращаемые значения всегда помещаются в регистре `AX` и флагах. Далее приводится пример ISR прерывания `13h`.

```

/*
  L2_13.C
  Фильтр прерывания 13h BIOS – управление диском.
  Устанавливает в значение YES внешнюю переменную
  inside_BIOS, когда активно прерывание 13h.
*/
#define NO 0
#define YES 1
#include <dos.h>
extern unsigned inside_BIOS;
extern void interrupt (*old_int13)(void);
void interrupt ISR_int13(unsigned bp, unsigned di,
                        unsigned si, unsigned ds,
                        unsigned es, unsigned dx,
                        unsigned cx, unsigned bx,
                        unsigned out_ax, unsigned ip,
                        unsigned cs, unsigned out_flags)
{
  inside_BIOS = YES;
  old_int13();
  disable(); /*при выходе из ISR флаги восстанавливаются
              из стека */
  out_ax = _AX; /* передача информации, */
                /* поступающей из BIOSa */
  asm { /* во внутренних регистрах */
        pushf
        pop out_flags;
      }
  inside_BIOS = NO;
}

```


В случае, когда запись данных на диск из резидентной программы не может начаться немедленно, данные помещают во внутренний буфер. Так как резидентная программа имеет весьма ограниченный объем памяти, приходится использовать дополнительные средства “ускорения” разгрузки буфера. Наиболее общее решение — использовать часто вызываемые прикладными программами прерывания, например прерывание от таймера.

Другая проблема — создать при доступе к файлам условия работы MS-DOS и функций Turbo C, полностью идентичных тем, которые имеют место при обычной, нерезидентной работе. Для функций префиксного файлового доступа обязательно переключение PSP (см. п.2.3.2). Если резидентная программа использует непосредственно либо через функции Turbo C FCB-функции MS-DOS, дополнительно требуется переключение *области переноса данных* (DTA — Data Transfer Area). С DTA работают функции просмотра директория `findfirst()`, `findnext()` и некоторые другие. MS-DOS поддерживает две специальные функции для упрощения работы по переключению DTA: MS-DOS-функция `AH = 2Fh` возвращает в `ES:BX` far-указатель на начало установленной в текущий момент DTA; MS-DOS-функция `AH = 21h` устанавливает новый адрес DTA, задаваемый парой регистров `DS:DX`. На этих MS-DOS-функциях базируются функции библиотеки Turbo C — `getdta()` и `setdta()`.

Приводимый далее фрагмент Си-подобного текста иллюстрирует последовательность действий по переключению DTA, выполняемых в случае, когда TSR работает с дисками:

```
/* Файл MAIN.C */
...
char far * old_DTA;      /* адрес “старой” DTA */
char far * TC_DTA; /* адрес DTA, используемой Turbo C */
...
main()
{
    ...
    TC_DTA = getdta(); /* сохраняем адрес DTA Turbo C */
}
```



```

    ...
}

/* Файл TSR_ACT.C */
...
extern char far * old_DTA;
extern char far * TC_DTA;
...
void TSR_prepare_activate(void)
{
    ...
    old_DTA = getdta(); /* сохранение текущей DTA      */
    setdta(TC_DTA);      /* переключение на DTA Turbo C */
    ...
    TSR_body();          /* обычная Си-функция        */
    ...
    setdta(old_DTA) .    /* обратное переключение DTA  */
    ...
}

```

Одним из эффективных решений при построении TSR является своппинг. *Своппинг* — это перенос программы с диска в память, когда программа получает управление, и обратный перенос данных и кода, когда истекает время, отведенное для работы программы. Своппинг широко используется в мультипрограммных операционных системах и позволяет за счет активного использования диска организовать параллельное исполнение многих программ даже при небольших объемах памяти. Резидентные программы — это форма пассивного параллелизма. Применительно к TSR своппинг означает следующее. В памяти размещается не вся TSR, а лишь ее минимальное ядро — обработчики прерываний. Определив необходимость и возможность активизации, ядро “подгружает” в память недостающую часть кода с диска. В случае, когда для загружаемого кода не хватает памяти, любая часть ее содержимого копируется на диск и на ее место помещается код TSR. Всегда существует вероятность того, что отгруженная на диск память — это фрагмент какого-либо обработчика прерываний, код которого может понадобиться-

ся системе. Для того чтобы этого не происходило, перед своппингом TSR с диска маскируются аппаратные прерывания. После завершения TSR восстанавливает память.

2.3.5. Перехват прерывания критической ошибки

При возникновении ошибки, неустранимой средствами MS-DOS, последняя передает управление ISR прерывания 24h (подробнее об этом см. в 5.1 второй книги комплекса). Правильно спроектированная TSR должна подменять перед “всплытием” этот вектор на адрес собственной функции и восстанавливать вектор перед выходом из TSR. Другими словами, когда TSR активна, обязательно действует собственный обработчик критической ошибки, даже если он не выполняет никаких действий.

При построении собственного обработчика прерывания 24h каскадное включение не используется. Вся обработка ответа пользователя (например, на ошибку неготовности диска — Abort, Retry, Ignore, Fail) должна выполняться TSR-программой.

Приводимый далее Си-подобный код иллюстрирует работу по перехвату прерывания критической ошибки:

```
/* Файл MAIN.C */
...
void interrupt (* old_int24)(void); /* "старая" ISR 24h */
...
main()
{
    ...
}

/* Файл TSR_ACT.C */
...
extern void interrupt (* old_int24)(void);
extern void interrupt (* ISR_int24)(void);
...
void TSR_prepare_activate(void)
{
    ...
    old_int24 = getvect(0x24); /* сохранение текущего
```



```

                                вектора 24h*/
setvect(0x24, ISR_int24);/*установка собственной ISR*/
...
TSR_body();                  /* обычная Си-функция */
...
setvect(0x24, old_int24); /* обратное переключение */
...
}

```

В простейшем случае ISR_24() просто игнорирует ошибку или без какой-либо дополнительной обработки дает команду MS-DOS вернуться в прикладную программу. Ниже приводится соответствующий этому текст функции:

```

/*
  L2_14.C
  "Заглушка" прерывания 24h – Критическая ошибка MS-DOS.
  Передает AX = 3 – Возврат в прикладную программу с
  индикацией ошибки MS-DOS.
*/
void interrupt ISR_int24(unsigned bp, unsigned di,
                        unsigned si, unsigned ds,
                        unsigned es, unsigned dx,
                        unsigned cx, unsigned bx,
                        unsigned out_ax,
                        unsigned ip, unsigned cs,
                        unsigned out_flags)
{
  out_ax = 3;      /* передача информации в MS-DOS */
}

```

При необходимости ISR_24 может сама проинтерпретировать код ошибки (регистр AL) и возможные ответы пользователя на нее (AH), вывести соответствующее приглашение и принять ответ. Безопасными для TSR будут ответы Fail (out_ax = 3) и Retry (out_ax = 1). Передача в MS-DOS ответа Abort (out_ax = 2) запрещается. В этом случае MS-DOS завершит TSR, которая к этому не готова, так как не восстановлена таблица векторов прерываний.

2.3.6. Перехват прерывания по нажатию клавиш Ctrl-Break

При обнаружения нажатия комбинации клавиш Ctrl-Break во время выполнения своих функций MS-DOS выдает прерывание 23h. По умолчанию обработчик этого прерывания немедленно завершает исполнение активной программы. Поэтому нажатие клавиш Ctrl-Break в то время, когда TSR активна, может вызвать ее завершение с невосстановленной таблицей векторов прерывания. Для того чтобы избежать этого, можно, как это было показано в предыдущем пункте, перехватывать прерывание 23h на время активности TSR. Существует более простой способ, не требующий перехвата прерывания. Перед активизацией резидентной программы реакция на нажатие клавиш Ctrl-Break устанавливается в состояние OFF. В результате MS-DOS будет проверять наличие нажатия клавиш Ctrl-Break только при выполнении функций ввода-вывода, связанных с файлами stdin, stdout и stderr. Но если TSR не использует эти операции, то и не ощущается нажатие клавиш Ctrl-Break. Перед завершением TSR реакция на нажатие клавиш Ctrl-Break восстанавливается. Для сохранения(восстановления) реакции на нажатие клавиш Ctrl-Break используется функция AH = 33h. Подфункция AL = 0 запрашивает текущую реакцию, возвращаемую в DL (0 — OFF, 1 — ON); подфункция AL = 1 устанавливает реакцию, заданную в DL. Данные подфункции имеют аналоги в библиотеке Turbo C getchbrk() и setcbrk().

Приводимый далее Си-подобный код иллюстрирует установку и восстановление реакции на нажатие клавиш Ctrl-Break.

```
/* Файл TSR_ACT.C */
...
int old_cbrk;
...
void TSR_prepare_activate(void)
{
    ...
    old_cbrk = getchbrk(); /* сохранение текущей реакции */
}
```



```

    setcbrk(0);          /* установка CTRL-BREAK = OFF */
    ...
    TSR_body();          /* обычная Си-функция          */
    ...
    setcbrk(old_cbrk);   /* восстановление реакции      */
    ...
}

```

2.3.7. Использование прерывания таймера

Персональный компьютер имеет службу реального времени, которую образует аппаратный таймер, генерирующий запрос аппаратного прерывания в среднем 18,2 раза в секунду. Системный таймер подключается к линии запросов IRQ0. Контроллер прерываний при получении запроса передает в процессор номер прерывания 8. Это прерывание обрабатывается ISR BIOS. Кроме разного рода системной работы, обработчик выполняет и инструкцию INT 1Ch, т.е. компьютер 18,2 раза в секунду вызывает ISR прерывания 1Ch. Системный обработчик этого прерывания состоит из единственной инструкции возврата из прерывания IRET. Другими словами, это программная “заглушка” для подключения собственных ISR. Помимо обычных действий, принятых для обработчиков прерываний (сохранение(восстановление) регистров, определение необходимости и возможности активизации TSR, активизация TSR), ISR прерывания 1Ch должна выдавать в программируемый контроллер прерываний команду EOI (подробнее об этом см. в 1.3 второй книги комплекса). Наиболее общее решение этого вопроса состоит в записи в порт 20h байта 20h (так называемое общее завершение).

Прерывание от таймера при построении резидентных программ используется, в основном, в следующих случаях:

1) построение TSR, активизируемых при наступлении какого-либо события в реальном времени. Частный случай таких программ — различного рода “часы” на экране или “записные книжки” с предупреждением о наступлении того или иного события. Во многих пособиях по Си приводятся подобные примеры;

2) TSR, реализующие “отложенные” по соображениям безопасности запросы активизации. Например, приведенная на рис. 2.4 схема построения резидентной части имеет один, не очевидный на первый взгляд недостаток: прерывание 28h генерируется, во многих приложениях недостаточно часто для нормального функционирования резидентной программы.

2.4. Удаление резидентной программы из памяти

Во многих случаях удобно оснастить резидентную программу средствами “чистого” завершения с восстановлением всех используемых системных ресурсов и освобождением занимаемой памяти. Возможны две различные схемы, отличающиеся друг от друга способами информирования TSR о необходимости завершить свое существование. Первый из них требует повторного запуска TSR-программы на исполнение со специальным ключом в командной строке. Функция `main()` проверяет, установлена ли уже TSR в памяти, и, если она там есть, вызывает ISR, используемую для предотвращения повторной установки, например ISR прерывания 2Fh (см. п. 2.2.3), со специальным значением в регистре AL. “Фильтр” прерывания 2Fh распознает команду на отгрузку “резидента” из памяти и выполняет все необходимые для этого действия. Другой способ — передача с клавиатуры команды на отгрузку в момент, когда TSR активизирована. В обоих случаях действия TSR по отгрузке из памяти одни и те же.

Функция отгрузки TSR из памяти должна обязательно выполнить:

- 1) восстановление таблицы векторов прерывания, что устраняет в дальнейшем возможность активизации TSR;
- 2) закрытие всех файлов программы;
- 3) освобождение памяти, занимаемой TSR;

4) возврат управления в прерванную программу.

Кратко прокомментируем особенности выполняемых шагов. При восстановлении таблицы векторов используются “старые” векторы, сохраненные при запуске TSR. Но после этого могли быть запущены и другие резидентные программы (“захватчики”), установившие свои векторы и переместившие завершаемую ISR из начала каскада обработчиков. Восстановление вектора в такой ситуации успешно “утопит” все резидентные программы, включенные в каскад раньше завершаемой программы. Но это только одна проблема. Если после этого “захватчик” сам завершится, он может установить в таблицу векторов векторы уже отсутствующей в памяти ISR. В результате неизбежно “повисание” системы. Поэтому, прежде чем восстанавливать таблицу векторов, выгружаемая TSR должна проверить, является ли она первой в каскаде ISR, либо уже “перцеплена” глубже по цепочке. Критерием является совпадение адреса точки входа в ISR со значением вектора прерывания, записанным в текущий момент в таблице. При несовпадении значений восстановление векторов и удаление TSR не выполняется. Как вариант, в такой ситуации можно выполнить “усыпление” TSR — запись в дополнительную внешнюю переменную заранее установленного значения. “Усыпление” требует доработки ISR — введения дополнительной проверки внешнего флага. Если он, например, YES, ISR сразу вызывает следующий обработчик в каскаде, не выполняя своих обычных функций по определению момента активизации TSR.

Если TSR является первой в цепочке обработчиков, проводится очистка блоков памяти, занятых TSR. При этом возможно возникновение фрагментации памяти, если после отгружаемой TSR устанавливались другие TSR. Для освобождения памяти используется MS-DOS-функция $AH = 49h$ или ее Turbo C аналог `freemem()`. Напомним, что Си-программа владеет двумя блоками памяти: блоком, в котором хранятся резидентные код и данные, и блоком, в котором хранятся строки среды программы (подробнее об этом см. в гл. 6 второй книги комплекса).

Первый блок начинается с параграфа `_psp`, второй — с параграфа, записанного по смещению `2Ch` в `PSP`.

Обычно открытые файлы закрываются автоматически программой-загрузчиком системы программирования на Си (см. 6.2 второй книги комплекса) при выполнении функций завершения. Так как отгрузка `TSR` выполняется нестандартно, закрытие файлов должно быть выполнено явно. В противном случае не будут освобождены элементы системной таблицы описаний открытых файлов (см. 2.7 второй книги комплекса). Не будет лишним закрыть не только открытые регулярные файлы на диске, но и файлы стандартного ввода-вывода с префиксами `0` — `4`. Напомним, что закрытие файлов требует переключения `PSP` резидентной программой. Если это сделано, можно использовать функции библиотеки `Turbo C`, в частности `close()` или `_close()`.

После того, как все действия по отгрузке выполнены, `TSR` возвращает управление в фоновую программу. Это безопасный способ завершения. Нельзя завершать `TSR` функцией `exit()` даже на переключенном `PSP`.

И еще одна рекомендация по отгрузке резидентной программы из памяти. Функция, отгружающая `TSR`, вызывается после исполнения всех шагов по подготовке привычной для Си-функций среды: переключения `PSP`, области переноса `DTA`, установленной реакции на нажатие клавиш `Ctrl-Break` и т.д. Таким образом, функцию отгрузки следует вызывать из пределов функции `TSR_prepare_activate()`. Для сигнализации о необходимости отгрузки используется внешний флаг, например по имени `unload`. Его установку выполняет `TSR`, обнаружив запрос на отгрузку.

Далее приводится текст функции `TSR_exit()`, выполняющей завершение резидентной программы, при установке которой перехватывались векторы прерываний `5h`, `13h`, `1Ch`, `28h` и `2Fh`. Функция вызывается из пределов обработчика прерывания `28h`.

```
/* L2_15.C
```

```
Выполняет отгрузку резидентной программы.
```

```
В о з в р а т :
```



```

NO (0) – отгрузка TSR в данный момент невозможна;
YES(1) – отгрузка TSR выполнена успешно.
*/
#include <dos.h>
#include <io.h>
#define NO      0
#define YES     1
/*"Старые" векторы прерываний,
   сохраненные инсталлятором.*/
extern void interrupt (*old_int5)(void);
extern void interrupt (*old_int1C)(void);
extern void interrupt (*old_int13)(void);
extern void interrupt (*old_int28)(void);
extern void interrupt (*old_int2F)(void);
/* Прототипы устанавливаемых ISR. */
void interrupt ISR_int5 (void);
void interrupt ISR_int1C(void);
void interrupt ISR_int13(void);
void interrupt ISR_int28(void);
void interrupt ISR_int2F(void);
extern unsigned _psp;
int TSR_exit(void)
{
    register env_seg, handle = 0;
    disable();
    /* Проверка возможности безопасной отгрузки TSR. */
    if(getvect(5)      != ISR_int5 ||
       getvect(0x13) != ISR_int13 ||
       getvect(0x1C) != ISR_int1C ||
       getvect(0x28) != ISR_int28 ||
       getvect(0x2F) != ISR_int2F)
        return NO; /* отгрузка в данный момент опасна
                     для системы */
    setvect(0x5,  old_int5 );
    setvect(0x13, old_int13);
    setvect(0x1C, old_int1C);
    setvect(0x28, old_int28);
    setvect(0x2F, old_int2F);
    /* Освобождение памяти, занятой TSR. */
    freemem(_psp);
    env_seg = *(unsigned far *)MK_FP(_psp, 0x2c);
    freemem(env_seg);
}

```



```

/* Заккрытие открытых файлов. */
for(handle = 0; handle < 5; handle++)
    close(handle);
enable();
return YES;
}

```

2.5. Резидентная программа записи текстового окна в файл

В данном параграфе рассматривается пример использования приведенных в 2.2 — 2.4 теоретических сведений о построении резидентной программы. Спецификация TSR копирования текстового окна в файл дана в п. 2.1.4. Основная идея построения TSR такова. Сама TSR выполняет две несходные задачи: отметку текстового окна на экране и запись содержимого окна в файл. Вся работа с экраном и клавиатурой выполняется функцией `TSR_screen_copу()`. Вызов этой функции происходит “немедленно” из пределов ISR прерывания `5h` без выполнения каких-либо проверок. Единственное требование заключается в необходимости переключения стека перед вызовом `TSR_screen_copу()`. Функция `TSR_screen_copу()` — это обычная Си-функция, которая может использовать все возможности Turbo C. Единственное ограничение связано с запретом на использование для ввода-вывода MS-DOS-функций. Практически это заставляет для ввода-вывода использовать обращение к BIOSу и непосредственный доступ к видеобufferу (функции консольного ввода-вывода Turbo C или собственные функции, рассмотренные в гл. 8 и 9 второй книги комплекса).

Завершив отметку копируемого текстового окна и получив команду на его сохранение в файле, функция `TSR_screen_copу()` принимает строку спецификации файла. Затем содержимое окна записывается во внутренний буфер памяти и в YES выставляется внешняя переменная `request`, сигнализирующая о необходимости отгрузки буфера. Если ISR

прерывания 5 вновь получает управление, а переменная `request` еще не сброшена в NO (внутренний буфер занят), вызов `TSR_screen_cory()` не выполняется и генерируется звуковой сигнал. Звуковой сигнал генерируется и в случае, когда `TSR_screen_cory()` застаёт экран в графическом режиме, на работу с которым приводимый вариант функции не рассчитан.

Если с клавиатуры принята команда на отгрузку TSR из памяти, `TSR_screen_cory()` устанавливает в YES внешнюю переменную `unload`. Отгрузку TSR из памяти выполняет функция `TSR_exit()`, вызываемая из функции `TSR_prepare_activate()`.

Запись содержимого внутреннего буфера выполняет функция `TSR_body()`. Так как работа с диском требует соблюдения многих предварительных условий, ее вызову предшествует вызов `TSR_prepare_activate()`. Функция `TSR_prepare_activate()` вызывается из ISR прерывания 28h или 1Ch. Прерывание от таймера используется для того, чтобы максимально ускорить отгрузку внутреннего буфера на диск или выгрузку TSR из памяти. Таким образом, рассматриваемая TSR копирования текстового окна строится по схеме, приведенной на рис. 2.5.

Сделаем некоторые пояснения к рисунку. Внутренняя статическая переменная `inside` блокирует повторное вхождение в функцию `ISR_screen_cory()`. Внешняя переменная `busy` используется для предотвращения повторного вхождения в функцию `TSR_prepare_activate()`, вызов которой возможен как из ISR прерываний 28h, так и ISR прерывания 1Ch. На рис. 2.5 не показаны “фильтры” прерываний 2Fh и 13h. Первый из них используется для предотвращения повторной установки TSR, а второй управляет флагом `inside_BIOS` (см. L2_13.C). Установку флагов `request` и `unload` выполняет функция `TSR_screen_cory`. Для большей надежности в рассматриваемой программе используются разные стеки: один применяется при активизации функции `TSR_screen_cory()` (“старые” значения регистров SS и SP запоминаются в переменных `old_screen_ss` и `old_screen_sp`, “новые” значения хранятся во внешних переменных `TSR_screen_ss` и `TSR_screen_sp`, а под стек

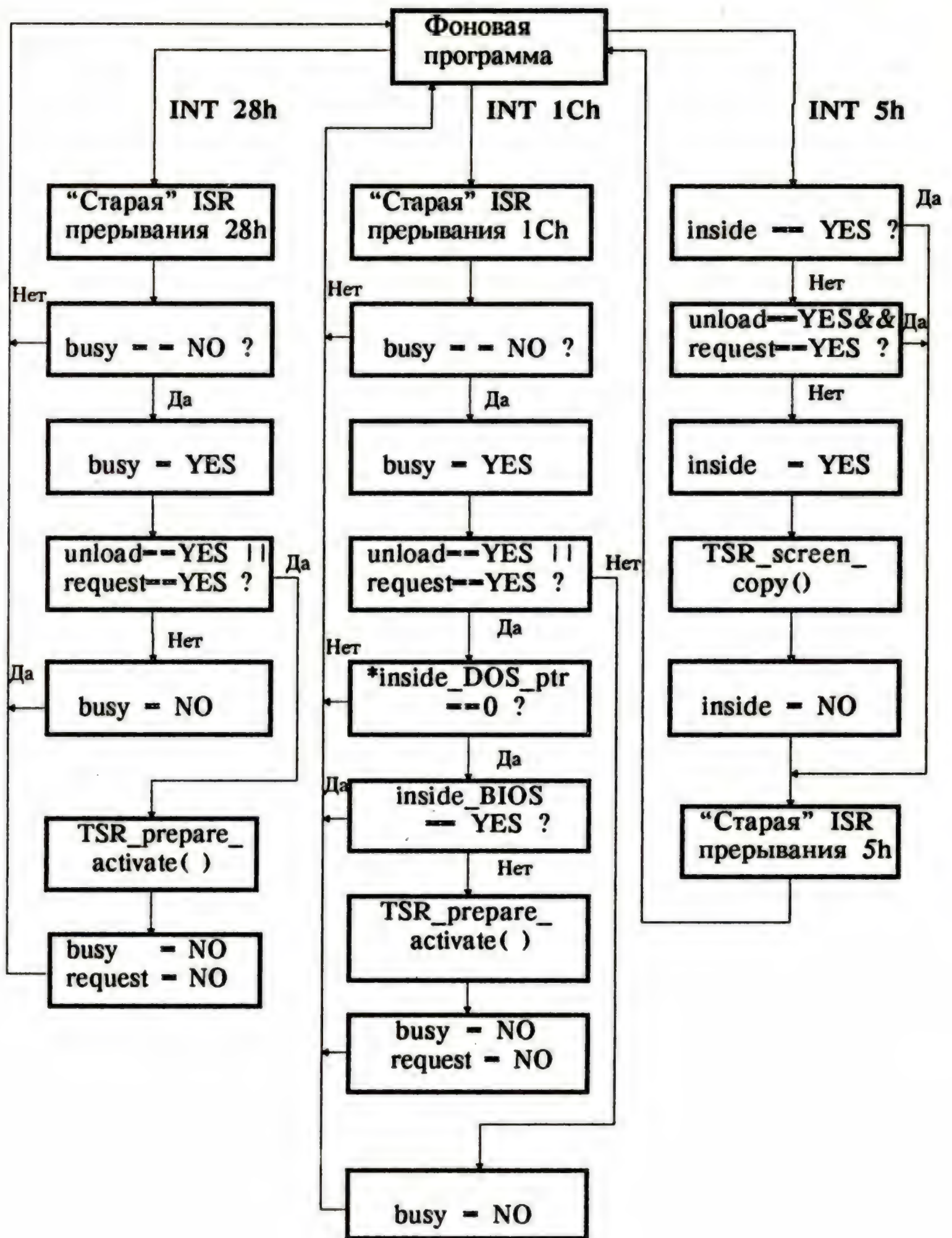


Рис. 2.5. Упрощенная схема взаимодействия обработчиков прерываний 1Ch (таймер), 28h, 5 (печать экрана)

используется массив `TSR_screen_stack[512]`), а второй используется функцией `TSR_body` ("старые" значения регистров `SS` и `SP` запоминаются в переменных `old_ss` и `old_sp`, "новые" значения хранятся во внешних переменных `TSR_ss` и `TSR_sp`, а под стек используется массив `TSR_stack[512]`).

Далее приводится текст функции `TSR_screen_copy()`:

```
/*
```

```
    L2_16.C
```

```
Задаёт инверсным атрибутом текстовое окно экрана. После
нажатия клавиши 'S' открывает окно, в котором задается
имя файла. При нажатии ENTER устанавливает внешнюю
переменную request в YES, а содержимое отмеченного
текстового окна помещает во внутренний буфер data[].
При нажатии клавиши 'Q' после активизации программы
устанавливает внешнюю переменную unload в YES.
Рассчитана на работу во всех цветных текстовых режимах.
```

```
*/
```

```
#include <dos.h>
```

```
#include "screen.h"
```

```
#include <bios.h>
```

```
#include <ctype.h>
```

```
#include <string.h>
```

```
#define ESC          27
```

```
#define SAVE          13
```

```
#define QUIT          51
```

```
#define OK             0
```

```
#define BAD_PARAMS    1
```

```
#define NO             0
```

```
#define YES            1
```

```
extern unsigned request, unload; /*флаги для
                                ISR_int28() и ISR_int1C() */
```

```
/* Внешние переменные,
```

```
используемые только в файле L2_16.C. */
```

```
unsigned str0 = 0, str1 = 10, col0 = 0, col1 = 10;
```

```
unsigned vid_memory = 0xB800, start_adr,
```

```
        max_str, max_stolb;
```

```
char filename[40], data[4100];
```

```
unsigned buf[120];
```



```

/* Прототипы используемых функций. */
int TSR_window(void);
    /* отмечает окно и принимает команду */
void inv_window(void);
    /* инвертирует атрибуты окна */
void get_window(void);
    /* записывает текст окна в буфер data */
void vert_inv(unsigned col);
    /* инвертирует атрибуты строки вертикально */
void hor_inv(unsigned str);
    /* инвертирует атрибуты строки горизонтально */
int video_params(void);
    /* определяет текущие параметры видеосистемы */
int get_string(unsigned str, unsigned col, char * ptr);
    /* прием строки с клавиатуры с "эхом" в нужном месте
    экрана */
void get_text(int, int, int, int, unsigned *);
    /* аналог gettext() Turbo C */
void put_text(int, int, int, int, unsigned *);
    /* аналог puttext() Turbo C */
void beep(int tone, int duration);
    /* сигнал частоты tone длительностью duration */

void TSR_screen_copy(void)
{ register ret;
  ret = TSR_window(); /* выполняет основную работу */
  switch(ret) {
  case SAVE:
      get_window(); /* запись окна в буфер */
      disable(); /* прерывания запрещены до
                  выхода из ISR_int5() */
      request = YES;
      return;
  case QUIT:
      unload = YES;
      return;
  case ESC:
      return;
  case BAD_PARAMS:
      beep(800, 500); /* сигнал 800 Гц длительностью 0.5 с */
  }
}

```


/*

TSR_WINDOW()

Отмечает инверсным атрибутом текстовое окно экрана; реагирует на клавиши Left, Right, Up, Down, Ctrl-Left, Ctrl-Right, Ctrl-Up, Ctrl-Down. При нажатии ESC и 'Q' восстанавливает атрибуты окна и завершается. При нажатии клавиши 'S' открывает окно, в котором задается имя файла.

В о з в р а т :

ESC (27) – нажата клавиша Esc;
SAVE (13) – принято имя файла и нажата
 клавиша ENTER;
QUIT (51) – нажата клавиша 'Q';
BAD_PARAMS(1) – неподходящий видеорежим.

*/

int TSR_window(void)

{

 register ret;

 union KEY {

 int i;

 char c[2];

 } key;

 static char * ftitle = "Имя файла назначения";

 static char * fkey = "ESC <Cancel> ENTER <OK>";

 static char style[8] = {'\xC9', '\xCD', '\xBB', '\xBA',
 '\xBA', '\xC8', '\xCD', '\xBC'};

 if(video_params() == BAD_PARAMS)

 return BAD_PARAMS; /* функция не рассчитана на
 эти случаи */

 cursor(OFF); /* выключаем курсор с текстового экрана */

/*

 Коррекция статических координат окна, если они
 превосходят текущие возможности экрана.

*/

 if(str1 > max_str)

 str1 = max_str;

 if(col1 > max_stolb - 1)

 col1 = max_stolb - 1;

 if(str0 > max_str)

 str0 = max_str;

 if(col0 > max_stolb - 1)

 col0 = max_stolb - 1;


```

inv_window(); /* координаты окна – статические
                                     внешние */
/* Бесконечный цикл приема и
   интерпретации ввода с клавиатуры. */
for(;;)
{
    key.i=bioskey(0);
    switch(key.c[0])
    {
        case 0: /* нажата специальная клавиша */
            switch(key.c[1])
            {
                case 75: /* Left */
                    if(col0)
                        { /* "перекраска" вертикальной полосы */
                            col0;
                            vert_inv(col0);
                        } break;

                case 77: /* Right */
                    if(col0 < col1)
                        { /* восстановление вертикальной полосы */
                            vert_inv(col0);
                            col0++;
                        } break;

                case 72: /* Up */
                    if(str0)
                        { /* "перекраска" горизонтальной полосы */
                            str0;
                            hor_inv(str0);
                        } break;

                case 80: /* Down */
                    if(str0 < str1)
                    {
                        /* восстановление горизонтальной полосы */
                        hor_inv(str0);
                        str0++;
                    } break;
            }
    }
}

```



```

case 115:          /* Ctrl-Left */
    if(col0 < col1)
        { /*восстановление вертикальной полосы*/
            vert_inv(col1);
            col1;
        } break;
case 116:          /* Ctrl-Right */
    if(col1 < max_stolb-1)
        { /* "перекраска" вертикальной полосы */
            col1++;
            vert_inv(col1);
        } break;
case 132:          /* Ctrl-PgUp */
    if(str0 < str1)
    {
        /* восстановление горизонтальной полосы */
        hor_inv(str1);
        str1;
    } break;
case 118:          /* Ctrl-PgDn */
    if(str1 < max_str)
        { /*"перекраска" горизонтальной полосы*/
            str1++;
            hor_inv(str1);
        } break;
} break;

default:          /* если нажата символьная клавиша */
switch(key.c[0])
{
    case 27:
        inv_window(); /* восстановление
                        атрибутов окна */
        cursor(ON); /* восстановление
                     курсора на экране */
        return(ESC);
    case 'q':
    case 'Q':
        inv_window(); /* восстановление
                        атрибутов окна */
        cursor(ON); /* восстановление
                     курсора на экране */

```



```

        return(QUIT);
    case 's':
    case 'S':
        get_text(0,0,39,2,buf); /* сохранение
                                   окна экрана */
        expl_win(0,0,2,39,0x1f, 0, 0x1f); /*
                                   "взрыв" окна */
        border(0, 0, 2, 39, ftitle,
                0x71, style, 0x1f);
        hor_prn(2, 8, fkey, 0x1f); /* строка
                                   внизу окна диалога */
        /* прием имени файла */
        ret = get_string(1, 1, filename);
        switch(ret)
        { case 27:
            /* Esc при наборе имени файла */
            /* восстановление экрана */
            put_text(0, 0, 39, 2, buf);
            break;
          case 13:
            /* Enter при наборе имени файла */
            /* восстановление экрана */
            put_text(0, 0, 39, 2, buf);
            /* восстановление атрибутов окна */
            inv_window();
            /* восстановление курсора на экране */
            cursor(ON);
            return (SAVE);
          }
        }
    }
} /* Завершение цикла приема-интерпретации
   ввода с клавиатуры. */

/*
   GET_WINDOW()
   Функция записи текстового окна, координаты которого
   заданы внешними переменными str0, col0, str1, col1, во
   внешний буфер data[].
   */
void get_window(void)

```



```

{
    char far * screen;
    int handle, row, col, ind, ret;
    /* Цикл "откачки" данных из видеопамати.
       В конце каждой строки экрана добавляются
       символы 0xd, 0xa. */
    ind=0;
    for(row = str0; row <= str1; row++)
    {
        screen = MK_FP(vid_memory, start_adr +
                        (row * max_stolb + col0) * 2);
        for(col = col0; col <= col1; col++, screen += 2)
            data[ind++] = * screen;
        data[ind++] = 0xD;
        data[ind++] = 0xA;
    }
    return;
}

```

```

/*
    VIDEO_PARAMS()
    Определение текущих параметров видеосистемы,
    используемых функциями вывода на экран.
    В о з в р а т :
        BAD_PARAMS (1) - установлен неподходящий режим;
        ОК          (0) - текущие параметры установлены
                        во внешние переменные.
*/

```

```

*/
int video_params(void)
{
    register int cur_mode;
    cur_mode = * (char far *)MK_FP(0x40, 0x49);
    /* Коррекция адреса начала видеопамати
       для адаптера монохром.*/
    if(cur_mode == 7)
        vid_memory = 0xb000;
    else
        vid_memory = 0xb800;
    /* Исключение работы в графическом режиме. */
    if(!(cur_mode == 7 | (cur_mode <= 3)))
        return BAD_PARAMS;
    max_str = * (char far *)MK_FP(0x40, 0x84);
}

```



```

    if(max_str == 0)    /* скорее всего, это 25 строк */
        max_str = 24;
    max_stolb = * (char far *)MK_FP(0x40, 0x4a);
    start_adr = * (unsigned far *)MK_FP(0x40, 0x4e);
    return OK;
}

/*
    HOR_INV()
Внутренняя функция, инвертирующая 7 младших бит
атрибута горизонтальной строки str в пределах
вертикальных координат, заданных внешними переменными
col0, col1.
*/
void hor_inv(unsigned string)
{
    extern unsigned str0, str1, col0, col1, vid_memory,
                    start_adr, max_str, max_stolb;
    unsigned j;
    unsigned int far * screen;
    screen = MK_FP(vid_memory,
                    start_adr + (string * max_stolb + col0) * 2);
    for(j = col0; j <= col1; j ++, screen ++)
        * screen = (* screen) ^ 0x7F00;
}

/*
    VERT_INV()
Внутренняя функция, инвертирующая 7 младших бит
атрибута вертикальной строки column в пределах
горизонтальных координат, заданных внешними переменными
str0, str1.
*/
void vert_inv(unsigned column)
{
    extern unsigned str0, str1, col0, col1, vid_memory,
                    start_adr, max_str, max_stolb;
    unsigned i;
    unsigned int far * screen;
    screen = MK_FP(vid_memory,
                    start_adr + (str0 * max_stolb + column) * 2);
    for(i = str0; i <= str1; i ++, screen += max_stolb)

```



```

    * screen = (* screen) ^ 0x7F00;
}

/*
    INV_WINDOW()
Внутренняя функция, инвертирующая 7 младших бит
атрибута прямоугольного окна экрана, заданного внешними
координатами str0, str1, col0, col1.
*/
void inv_window(void)
{
    extern unsigned str0, str1;
    unsigned i;
    for(i = str0; i <= str1; i++)
        hor_inv(i);
}

/*
    GET_STRING()
Внутренняя функция приема строки с клавиатуры и
эхоирования ввода на экране. "Эхо" начинается с точки
string, column. Строка записывается в массив fbuf. Имя
файла – статическое.
В о з в р а т :
    ESC (27) – нажата клавиша Esc;
    SAVE (13) – нажата клавиша ENTER.
*/
int get_string(unsigned string, unsigned column,
               char * fbuf)
{
    extern unsigned vid_memory, start_adr;
    unsigned int far * screen, j;
    union {
        int i;
        char c[2];
    } key;

    hor_prn(string, column, fbuf, 0x1e);
    j = strlen(fbuf);
    fbuf += j;
    j += column;
    screen = MK_FP(vid_memory,

```



```

        start_adr + (string * max_stolb + j) * 2);
for(;;) /* бесконечный цикл приема
        символов с клавиатуры */
{
    key.i=bioskey(0);
    if(key.c[0])
    {
        switch(key.c[0]) {
        case 13:
            * fbuf = 0;
            return SAVE;
        case 27:
            return ESC;
        case 8:      /* Backspace */
            if(j > column)
            {
                fbuf;
                j;
                screen ;
                * screen = (* screen) & 0xFF00;
            } break;
        default:
            if(j < 39)
            {
                key.c[0] = toupper(key.c[0]);
                * fbuf++ = key.c[0];
                * screen = 0x1e00 | key.c[0];
                j ++; screen ++;
            } }
        }
    }
}

```

/*

GET_TEXT.C

Внутренняя функция сохранения окна экрана, левый верхний угол которого c0, s0, а правый нижний – c1, s1, в буфер buf. Функция Turbo C gettext() не подходит, так как рассчитана на случай не более 25 строк на экране.

*/

```

void get_text(int c0, int s0, int c1, int s1,
              unsigned * buf)

```



```

{
    unsigned far * screen;
    int row, col;
    for(row = s0; row <= s1; row++)
    {
        screen = MK_FP(vid_memory,
            start_adr + (row * max_stolb + c0) * 2);
        for(col = c0; col <= c1; col++)
            * buf++ = * screen++;
    }
}

```

/*

PUT_TEXT.C

Внутренняя функция восстановления окна экрана, левый верхний угол которого c0, s0, а правый нижний – c1, s1, из буфера buf. Функция Turbo C puttext() не подходит, так как рассчитана на случай не более 25 строк на экране.

*/

```

void put_text(int c0, int s0, int c1, int s1,
    unsigned *buf)
{
    unsigned far * screen;
    int row, col;
    for(row = s0; row <= s1; row++)
    {
        screen = MK_FP(vid_memory,
            start_adr + (row * max_stolb + c0) * 2);
        for(col = c0; col <= c1; col++)
            * screen++ = *buf++;
    }
}

```

/* BEEP()

Внутренняя функция генерации звукового сигнала частоты tone длительностью duration миллисекунд.

*/

```

void beep(tone, duration)
{
    sound(tone);
    delay(duration);
    nosound();
}

```


Отметим одну особенность, присущую функции `TSR_screen_copy()`. Любой вывод на экран (сообщения или ввод) выполняется в окна с обязательным сохранением предыдущего содержимого экрана и его восстановлением после завершения операции, что необходимо в резидентных программах. В остальном это обычная Си-функция. Для того чтобы программа могла работать корректно для всех текстовых режимов, текущие параметры видеосистемы определяются при каждом вхождении в функцию `TSR_screen_copy()` динамически. Так как функция вызывается из обработчика аппаратного прерывания без каких-либо проверок, защищающих TSR от повторного вхождения в MS-DOS, ввод-вывод должен выполняться только непосредственным доступом к экрану и клавиатуре или средствами BIOS. Перед вызовом функции выполняется переключение стека. Вызов `TSR_screen_copy()` из пределов резидентного кода выполняет обработчик прерывания 5, текст которого приведен ниже. Его структура соответствует схеме, приведенной на рис. 2.5.

```
/*
```

```
    L2_17.C
```

```
ISR прерывания 5. Вызывает, предварительно переключив
стек, функцию TSR_screen_copy(). Если внешние
переменные request и unload равны YES, генерирует
звуковой сигнал и сразу вызывает "старый" обработчик.
Статическая переменная inside блокирует повторное
вхождение в ISR.
```

```
*/
```

```
#include <dos.h>
```

```
#define      YES  1
```

```
#define      NO   0
```

```
extern void interrupt (*old_int5)(void);
```

```
extern unsigned old_screen_ss, old_screen_sp,
```

```
            TSR_screen_ss, TSR_screen_sp, request, unload;
```

```
static unsigned inside;
```

```
void TSR_screen_copy(void); /* прототип
```

```
вызываемой функции */
```

```
void beep(int, int); /* генерация звукового сигнала
                      (см. L2_16.C) */
```

```
void interrupt ISR_int5(void)
```



```

{
    if(inside == NO)
    {
        inside = YES;
        if(request != YES && unload != YES)
        {
            /* Переключение стека для функции
               TSR_screen_copy(). */
            old_screen_ss = _SS;
            old_screen_sp = _SP;
            disable();      /* установку новых SS,SP
                           не прерывать */
            _SS = TSR_screen_ss;
            _SP = TSR_screen_sp;
            enable();
            TSR_screen_copy();
            /* Обратное переключение стека. */
            disable();      /* установку новых SS,SP
                           не прерывать */
            _SS = old_screen_ss;
            _SP = old_screen_sp;
        }
        else
            beep(1000, 1000); /* сигнал 1000 Гц
                               длительностью 1 с */

        inside = NO;
    }
    old_int5(); /* вызов "старого"
                обработчика прерывания 5 */
}

```

Обработчик прерывания 28h, приведенный в файле L2_12.C, потребует некоторой доработки, связанной с организацией взаимодействия с обработчиком прерывания 1Ch через внешнюю переменную busy и необходимостью отгрузки резидентной программы из памяти (см. 2.4). Далее приводится пример обработчика прерываний 28h, переработанного в соответствии со схемой, данной на рис. 2.5.

```

/*
   L2_18.C
   ISR прерывания 28h: если имеются "отложенные" попытки
   активизации или деактивизации TSR, вызывает функцию

```



```

TSR_prepare_activate().
*/
#define      NO      0
#define      YES     1

#include <dos.h>
extern void interrupt (*old_int28)(void);
extern unsigned busy, request, unload;
void TSR_prepare_activate(void); /* готовит активизацию
                                TSR_body() */
void interrupt ISR_int28(void)
{
    old_int28(); /* вызов "старой" ISR прерывания 28h */
    disable();   /* маскирование аппаратных прерываний */
    if(busy == NO)
    {
        busy = YES;
        if(request == YES || unload == YES)
            TSR_prepare_activate(); /* активизация TSR */
        request = NO;
        busy = NO;
    }
    return;
}

```

Вызываемая из обработчиков прерываний 1Ch и 28h функция `TSR_prepare_activate()` создает привычную для Си-функций среду. Это необходимо для правильной работы с файлами, которую выполняют функции записи текстового окна в файл `TSR_body()`, и отгрузки TSR из памяти `TSR_exit()`. В случае, когда отгрузка TSR из памяти не может быть выполнена, генерируется звуковой сигнал. Далее приводится текст функции `TSR_prepare_activate()`:

```

/*
    L2_19.C    -   Файл TSR_ACT.C
    Готовит активизацию резидентной функции: переключает
    стек, область переноса данных и PSP, устанавливает
    временную реакцию на критическую ошибку и нажатие
    клавиш Ctrl-Break. Если внешняя переменная request =
    =YES, вызывает TSR_body(). Если внешняя переменная
    unload = YES, отгружает TSR из памяти функцией

```



```

TSR_exit(). Затем восстанавливает систему в прежнее
состояние (до вызова TSR). */
#define      NO      0
#define      YES     1
#include <dos.h>
extern unsigned TSR_ss, TSR_sp, old_ss, old_sp;
extern unsigned _psp, old_PSP;
extern unsigned request, unload;
extern char far * old_DTA;
extern char far * TC_DTA;
int old_cbrk;
extern void interrupt (* old_int24)(void);
void interrupt ISR_int24(void);
void beep(int, int); /* звуковой сигнал заданной частоты
                        и длительности */
int TSR_exit(void); /* выполняет отгрузку TSR из памяти
                     (см. L2_15.C) */
void TSR_body(void); /* обычная Си-функция */

void TSR_prepare_activate()
{
    /* Переключение стека. */
    disable();
    old_ss = _SS;
    old_sp = _SP;
    _SS = TSR_ss;
    _SP = TSR_sp;
    _BP = _SP;
    /* Переключение текущего PSP на PSP резидентной
       программы. */
    _AH = 0x62;
    geninterrupt(0x21);
    old_PSP = _BX;
    _AH = 0x50;
    _BX = _psp;
    geninterrupt(0x21);
    old_DTA = getdta(); /* сохранение текущей DTA */
    setdta(TC_DTA);     /* переключение на DTA Turbo C */
    old_int24 = getvect(0x24); /* сохранение текущего
                                вектора 24h*/
    setvect(0x24, ISR_int24); /* установка собственной ISR*/
    old_cbrk = getcbrk(); /* сохранение текущей реакции */
}

```



```

    setcbreak(0);          /* установка CTRL-BREAK = OFF */
    enable();              /* разрешение прерываний      */
    if(request == YES)
        TSR_body();
    if(unload == YES)
        if(TSR_exit() == NO)
        {
            beep(3000, 1500); /* сигнал 3000 Гц
                                длительностью 1.5 с*/
            unload = NO; /* запрос не может быть выполнен */
        }
    disable(); /* запрещение прерываний от аппаратуры */
    setcbreak(old_cbrk); /* восстановление реакции */
    setvect(0x24, old_int24); /* обратное переключение */
    setdta(old_DTA); /* обратное переключение DTA */
    /* Обратное переключение PSP
    на PSP фоновой программы. */
    _AH = 0x50;
    _BX = old_PSP;
    geninterrupt(0x21);
    /* Обратное переключение стека. */
    _SS = old_ss;
    _SP = old_sp;
    return;
}

```

В связи с тем, что функция `TSR_prepare_activate()` может вызываться из двух асинхронно взаимодействующих ISR, существует вероятность повторного вхождения в функцию. Ее устранением занимаются обработчики прерываний, использующие семафор `busy`. Кроме того, все операции по прямому и обратному переключению ресурсов выполняются на маскированных аппаратных прерываниях. Это несколько повышает надежность работы TSR.

Функция `TSR_body()` выполняет запись текстового буфера `data[]` в файл. Заполнение буфера и прием строки имени файла в массив `filename[40]` выполняет функция `TSR_screen_cory()`, вызываемая обработчиком прерывания 5. Функция `TSR_body()` — это обычная Си-функция. Ее особенностью является то, что вывод сообщений об ошибках доступа к файлу выполняется в окна с сохранением

предыдущего содержимого экрана и его дальнейшим восстановлением. Далее приводится текст функции TSR_body():

```
/*
    L2_20.C
    Функция записи текстового буфера data[] в файл,
    спецификация которого хранится во внешнем массиве
    filename[40].
*/
#include <dos.h>
#include <fcntl.h>
extern char filename [40], data[4100];
extern unsigned str0, str1, col0, col1;
void win_error(char *);
unsigned buf_err[200];

void TSR_body(void)
{
    int handle, ret, count;
    /* Попытка открытия файла и обработка ошибки. */
    if((handle= _open(filename, O_RDWR)) == - 1)
        if((handle= _creat(filename, FA_ARCH))== -1)
        {
            win_error("Ошибка открытия файла.");
            return;
        }
    /* Определение общего количества
       записываемых символов. */
    count = (str1 - str0 + 1) * (col1 - col0 + 3);
    /* Запись информации в файл. */
    lseek(handle, 0L, SEEK_END);
    ret = write(handle, data, count);
    if(ret != count || ret == - 1)
    {
        win_error("Ошибка при записи в файл");
        _close(handle);
        return;
    }
    _close(handle);
    return;
}
```


/*

WIN_ERROR()

Внутренняя функция вывода сообщения об ошибке, ее причине и имени файла. Вывод выполняется в окно экрана. Возврат управления происходит по нажатию любой клавиши.
*/

```
void win_error(char * msg)
```

```
{
```

```
    static char style[8] = {'\xDA', '\xC4', '\xB7', '\xB3',  
                             '\xBA', '\xD4', '\xCD', '\xBC'};
```

```
    get_text(0, 9, 39, 13, buf_err); /*сохранение экрана*/  
    expl_win(9, 0, 13, 39, 0x40, 0, 0x40); /*"взрыв" окна*/  
    border(9,0,13,39, "ЗАПИСЬ ФАЙЛА НА ДИСК",  
           0x4f, style, 0x4f);
```

```
    hor_prn(10, 1 + (38 - strlen(msg))/2, msg, 0x4f);
```

```
    hor_prn(11, 1, "Файл:", 0x4f);
```

```
    hor_prn(12, 1 + (38 - strlen(filename))/2,  
            filename, 0x4f);
```

```
    hor_prn(13, 8, " Нахмите любую клавишу ", 0x4e);
```

```
    bioskey(0); /* ожидание нажатия любой клавиши */
```

```
    put_text(0,9,39,13,buf_err); /* восстанволение экрана*/  
    return;
```

```
}
```

Для ускорения отгрузки информации из текстового буфера data[] используется прерывание от таймера. Активизация резидентной программы в этом случае требует наиболее тщательной подготовки. Необходимо дополнительно проанализировать, не является ли MS-DOS фоновой программой во время выполнения нереентерабельной секции кода (см. п.2.3.3) и не является ли фоновой программой BIOS во время выполнения прерывания 13h (см. п. 2.3.4). Далее приводится листинг ISR, включаемой инсталлятором в каскад обработчиков прерывания 1Ch. Основная его задача — увеличить число моментов вре-

мени, когда TSR_prepare_activate() получает управление, если настал момент ее активизации.

/*

L2_21.C

ISR прерывания от таймера. Анализирует внешние флаги request и unload и, если они равны YES, проверяет состояние BIOS и MS-DOS. Если фоновая программа – не BIOS-ISR прерывания 13h или не MS-DOS в нереентерабельной секции, вызывает функцию TSR_prepare_activate.

*/

#define NO 0

#define YES 1

#define OK 0

#include <dos.h>

extern void interrupt (*old_int1C)(void);

void TSR_prepare_activate(void); /* готовит активизацию
TSR_body() */

extern unsigned inside_BIOS, request, unload, busy;

extern char far * inside_DOS_ptr;

void interrupt ISR_int1C(void)

{

old_int1C(); /* “старая” ISR прерывания таймера */
disable();

if(busy == NO)

{

busy = YES;

if(request == YES || unload == YES)

{

if(*inside_DOS_ptr == 1) /* MS-DOS занята */

{

busy = NO;

return;

}

if(inside_BIOS == YES)

{

busy = NO;

return;

}

outportb(0x20, 0x20); /* выдача команды EOI в
контроллер прерываний */


```

        enable();
        TSR_prepare_activate(); /* активизация TSR */
        request = NO;
        busy = NO;
    }
    busy = NO;
}
return;
}

```

ISR прерывания таймера подобна ISR прерывания 28h, однако перед входением в TSR по прерыванию от таймера обязательно выполняется сброс запроса прерывания от таймера в специальной микросхеме — контроллере прерываний. Именно это действие выполняет строка кода

```
outportb(0x20, 0x20);
```

В противном случае система не будет реагировать на остальные запросы прерываний от аппаратуры, в том числе и связанные с нажатием клавиш клавиатуры, до тех пор, пока не завершится активность TSR. Как результат, TSR не сможет получать клавиатурный ввод, будут потеряны все последующие прерывания от таймера, что нарушит точность системных часов и пр.

И, наконец, приведем текст функции `main()`, выполняющей инсталляцию обработчиков прерываний и резидентное завершение. Дизайн таких функций обсуждался в 2.2.

```

/*
    L2_22.C
    Инициализирующая функция. Устанавливает резидентную
    программу копирования текстового окна экрана в файл на
    диске. Для активизации TSR используется прерывание
    печати экрана.
*/
#include <dos.h>
#include <stdio.h>
#define YES    1
#define NO     0

```



```

/* значения SS, SP для работы TSR */
unsigned TSR_ss, TSR_sp, TSR_screen_ss,
        TSR_screen_sp;
unsigned TSR_stack[512],          /* используются
        TSR_screen_stack[512]; под стеки TSR */
unsigned old_ss, old_sp, old_screen_ss, old_screen_sp;
        /* "старые" SS, SP при входе в TSR */
unsigned old_PSP; /* хранит PID фоновой программы */
unsigned inside_BIOS = NO, busy = NO,
        request = NO, unload = NO; /* флаги TSR */
char far * inside_DOS_ptr; /* адрес флага повторной
        входимости в MS-DOS */
char far * old_DTA; /* адрес "старой" DTA */
char far * TC_DTA /* адрес DTA, используемой Turbo C */
void interrupt (* old_int24)(void);
void interrupt (* old_int5 )(void);
void interrupt (* old_int13)(void);
void interrupt (* old_int1C)(void);
void interrupt (* old_int28)(void);
void interrupt (* old_int2F)(void);
/* Прототипы собственных функций. */
void interrupt ISR_int28(void); /* L2_18.C */
void interrupt ISR_int1C(void); /* L2_21.C */
void interrupt ISR_int5 (void); /* L2_17.C */
void interrupt ISR_int13(void); /* L2_13.C */
void interrupt ISR_int24(void); /* L2_14.C */
void interrupt ISR_int2F(void); /* L2_8.ASM */
void tsr(char status, unsigned size); /* L2_6.C */
int is_installed(void); /* L2_9.C */
void print_help(void); /* L2_10.C */
extern unsigned _heaplen = 1024; /* задание
        размера "кучи" */
extern unsigned _stklen = 512; /* задание
        размера стека */
int main(int argc, char ** argv)
{
    unsigned size;
    if(is_installed() == YES)
    {
        printf("\n\аПрограмма %s уже установлена.\n\n",
                argv[0]);
        print_help();
    }
}

```



```

    return 1;
}
/* Определение адреса флага
повторной входимости в MS-DOS. */
_AH = 0x34;
geninterrupt(0x21);
inside_DOS_ptr = MK_FP(_ES, _BX);
/* Инициализация переменных,
используемых при переключении стека. */
TSR_ss = FP_SEG((unsigned far *)TSR_stack);
TSR_sp = FP_OFF(TSR_stack) + 1020; /* стек "растет"
от старших адресов к младшим */
TSR_screen_ss =
    FP_SEG((unsigned far *)TSR_screen_stack);
TSR_screen_sp = FP_OFF(TSR_screen_stack)+1020;
/* Сохранение адреса DTA Turbo C. */
TC_DTA = getdta();
/* Сохранение "старых" и установка
"новых" векторов прерываний. */
old_int5 = getvect(0x5 );
old_int13 = getvect(0x13);
old_int1C = getvect(0x1C);
old_int28 = getvect(0x28);
old_int2F = getvect(0x2F);
/* Перенос "старого" вектора 2Fh
в кодовый сегмент ISR_int2F(). */
movedata(_DS, &old_int2F, _CS,
    FP_OFF(ISR_int2F) + 9, 4 );
disable();
setvect(0x5 , ISR_int5 );
setvect(0x13, ISR_int13);
setvect(0x1C, ISR_int1C);
setvect(0x28, ISR_int28);
setvect(0x2F, ISR_int2F);
enable();
/* Резидентное завершение программы. */
size = _SS + (_SP + 15) / 16 - _psp;
print_help();
tsr(0, size);
return 255; /*неисполняемый код */
}

```


Для получения загрузочного модуля резидентной программы копирования окна экрана на диск следует выполнить совместную компиляцию следующих файлов: L2_6.C, L2_8.ASM, L2_9.C, L2_10.C, L2_13.C — L2_22.C и группы файлов, тексты которых приведены во второй книге комплекса: L8_9.C, L9_4.C — L9_6.C, L9_9.C. Скомпонованная программа имеет небольшую длину (порядка 20K байт) и работает достаточно устойчиво “под” теми программами, которые не отключают при своей работе прерывание 5, например Norton Commander, IDE Borland C++ и др. В следующем параграфе обсуждаются особенности использования прерывания от клавиатуры для активизации TSR.

2.6. Резидентные программы, использующие прерывание от клавиатуры. “Горячие” клавиши

Для активизации TSR наиболее часто используется некоторая комбинация клавиш, называемая далее *“горячей” клавишей*. Для того чтобы TSR реагировала на “горячую” клавишу, приходится использовать собственную ISR прерывания 9. Как правило, собственные ISR этого прерывания включаются по каскадной схеме. Во-первых, нельзя отключить стандартный BIOS-обработчик прерывания 9, который выполняет множество самых разных функций в интересах всей системы (подробнее об этом см. в гл. 7 второй книги комплекса). Полная эмуляция его работы требует немалого труда, хотя и существует целый ряд пакетов, выполняющих эту работу, например Quick C, Microsoft C, SideKick Plus и др. Во-вторых, использование каскадного включения не “топит” другие TSR, “зацепленные” ранее за прерывание от клавиатуры. Выбранная “горячая” клавиша не должна

совпадать ни с одной из “горячих” клавиш других ISR каскада.

Общая схема построения ISR прерывания от клавиатуры подобна использованной для ISR прерывания 5 (L2_17.C, ISR_int5()). Но обработчик ISR_int9() дополнительно анализирует, наступил ли момент активизации. Он задается состоянием шифт- и триггерных клавиш, возможно, в сочетании со специальной или обычной символьной клавишей. Определение состояния шифт- и триггерных клавиш можно выполнять непосредственным доступом в память (см. 7.5 второй книги комплекса) или через прерывание 16h BIOS. Признаком нажатия нужной специальной или символьной клавиши будет появление в буфере клавиатуры нужного двухбайтового BIOS-кода клавиши. Из этих двух байтов лучше ориентироваться на старший, где помещается скэн-код клавиши. Еще более надежным будет чтение скэн-кода не из буфера клавиатуры, а непосредственно из порта программируемого периферийного интерфейса (порт 0x60). Чтение кода из буфера клавиатуры выполняется без его извлечения (т.е. без продвижения указателя “головы”). После того, как обнаружен BIOS-код клавиши, входящей в “горячую” комбинацию, код клавиши удаляется из буфера.

Далее приводится пример ISR прерывания 9, заменяющей ISR прерывания 5 в примере, рассмотренном в 2.5. Для активизации резидентной программы выбрана “экзотическая” комбинация LeftShift+Ctrl+S.

```
/*
```

```
  L2_23.C
```

```
Обработчик прерывания от клавиатуры. При одновременном
нажатии клавиш LeftShift+Ctrl+S вызывает, предвари-
тельно переключив стек, функцию TSR_screen_copy().
Если внешние переменные request и unload равны YES,
генерирует звуковой сигнал и сразу вызывает “старый”
обработчик. Статическая переменная inside блокирует
повторное входение в ISR. Заменяет файл L2_17.C в
файле проекта предыдущего параграфа.
```

```
*/
```



```
#include <dos.h>
#include <bios.h>
#define YES 1
#define NO 0
#define SHIFT_MASK 0x0006 /* маска шифт-клавиш */
#define SCAN_CODE '\x1F' /* скэн-код клавиши 'S' */

void TSR_screen_copy(void); /* прототип
                             вызываемой функции */
void beep(int, int); /*генерация звукового
                     сигнала (см. L2_16.C)*/

extern void interrupt (*old_int9)(void);
extern unsigned old_screen_ss, old_screen_sp,
               TSR_screen_ss, TSR_screen_sp, request, unload;
static unsigned inside;

void interrupt ISR_int9(void)
{
    old_int9(); /* вызов "старого"
                обработчика прерывания 9 */
    disable();
    if(inside == NO)
    {
        inside = YES;
        if(request != YES || unload != YES)
        {
            /* Проверка того, нажата ли "горячая" клавиша. */
            if((bioskey(2) & SHIFT_MASK) == SHIFT_MASK)
                if(inportb(0x60) == SCAN_CODE)
                {
                    bioskey(0); /* удаление символа 'S'
                                из буфера клавиатуры */
                }
            /*Переключение стека для функции TSR_screen_copy().*/
            old_screen_ss = _SS;
            old_screen_sp = _SP;
            disable(); /* установку новых SS,SP
                        не прерывать */
            _SS = TSR_screen_ss;
            _SP = TSR_screen_sp;
            enable();
            TSR_screen_copy();
        }
    }
}
```



```

        /* Обратное переключение стека. */
        disable(); /* установку новых SS,SP
                   не прерывать */
        _SS = old_screen_ss;
        _SP = old_screen_sp;
    }
}
else
    beep(1000, 1000); /* сигнал 1000 Гц
                      длительностью 1 с */
inside = NO;
}
}

```

Для включения рассмотренного обработчика в состав программы копирования текстового окна экрана (см. 2.5) необходимы некоторые изменения в функциях `main()` (L2_22.C) и `TSR_exit()` (L2_15.C). Всюду, где есть ссылка на `ISR_int5`, ее следует заменить ссылкой на `ISR_int9`, а ссылку на `old_int5` — ссылкой на переменную `old_int9`. В функциях `setvect()` номер вектора прерывания заменяется с 5 на 9. И, разумеется, в файле L2_10.C исправляется текст сообщения об активизации программы. Все остальное не требует изменения.

2.7. Резидентные программы, использующие прерывание 16h

Прерывание 16h — основной способ получения клавиатурного ввода прикладной программой и функциями MS-DOS. Если установлен собственный обработчик прерывания, дополняющий стандартный BIOS— обработчик прерывания 16h, TSR-программа будет активизироваться всякий раз, когда происходит обращение к буферу клавиатуры и определение состояния шифт- и триггерных кла-

виш. Дополнительные действия, выполняемые TSR-программой, могут быть самыми разными. Основная идея — передача через прерывание 16h дополнительной информации в прикладную программу. Эта информация может передаваться в программу непосредственно в регистрах, но тогда существует опасность того, что подход не сработает для тех программ, которые читают клавиатурный ввод непосредственно из буфера. Поэтому более общим будет подход, при котором дополнительная информация записывается в буфер клавиатуры, а оттуда считывается при выполнении прерывания 16h. Перечислим лишь некоторые из возможных дополнительных действий, выполняемых собственными обработчиками прерывания 16h:

1) демонстрационные программы записывают в буфер клавиатуры коды нажатий клавиш из специфицированного внешнего файла до тех пор, пока буфер не заполнится, либо пока не закончится файл, и возвращают управление в точку вызова прерывания 16h. В результате в прикладную программу поступают коды клавиш из буфера без фактического нажатия клавиш, но программа выполняет все действия так, как будто эти нажатия выполнены. Такие программы неоценимы при построении обучающих программ и демонстрации работы программ в режиме “автопилота”. К сожалению, рассмотренный подход не работает для программ, читающих ввод с клавиатуры непосредственно из порта 60h. Правда, число таких прикладных программ невелико. Для упрощения подготовки файлов, содержащих двухбайтовые коды нажатий клавиш, программы дополняются средствами автоматической регистрации выполняемых нажатий клавиш и записи кодов из буфера клавиатуры во внешний файл. Последовательность нажатий клавиш часто называют “сценарием” (*script*). Специальные “горячие” клавиши начинают и завершают запись сценария. Затем полученный сценарий будет “сыгран” на прикладной программе. Возможная опция поведения таких программ — введение

холостых циклов, замедляющих запись символов в буфер клавиатуры с тем, чтобы дать возможность пользователю успевать отслеживать все действия программы, работающей на “автопилоте”;

2) перенос из файлов на диске в произвольный текстовый редактор кусков текста, начиная с текущей позиции курсора. Так работают некоторые специальные Help-системы, в том числе и резидентная программа THELP.COM, входящая в стандартную поставку Turbo C. Программа после установки в памяти активизируется нажатием клавиши 5 на цифровой клавиатуре и выдает оперативную подсказку по слову, находящемуся в текущей позиции курсора точно так, как это было бы после нажатия Ctrl-F1 при работе в IDE. Если необходимо, окно информации Help может быть скопировано в редактор, из которого была активизирована THELP.COM. Идея построения таких программ состоит в следующем. Установленный обработчик прерывания 9 отслеживает “горячую” клавишу по скэн-коду с клавиатуры. Обнаружив, что активизация необходима, обработчик прерывания 9 вызывает функцию чтения слова с экрана. (Пример такой функции приведен в 9.6 второй книги комплекса; см. L2_14.C.) Определив нужное слово, TSR-программа записывает непосредственно в видеобуфер закрепленное за словом окно информации. Предыдущее содержимое окна сохраняется в промежуточном буфере. Обнаружив “горячую” клавишу, задающую перенос в редактор, TSR-программа символ за символом записывает текст окна в буфер клавиатуры;

3) упрощенным вариантом такой программы является программа преобразования слов информации в текстовых редакторах, например путем перевода с прописных букв в строчные или с латинских в русские (так называемые программы “*вращения*” регистров). Активизируясь по закрепленной “горячей” клавише или обнаруживая специальный код непосредственно в буфере клавиатуры, TSR-

программа читает слово в текущей позиции курсора. Затем она “стирает” преобразуемое слово из редактора. Для этого в буфер клавиатуры записывается нужное число кодов нажатий клавиш управления курсором так, чтобы курсор установился на границу слова, например правую, если для стирания будет “использоваться” клавиша BackSpace. Затем в буфер клавиатуры записывается код клавиши BackSpace столько раз, сколько букв в преобразуемом слове. Подавляющее большинство текстовых редакторов обрабатывают BackSpace одинаково — стирая символ в позиции слева от курсора. Поэтому TSR-программа “стирает” слово в редакторе. И, наконец, в буфер клавиатуры записываются символы преобразованного слова;

4) другой вариант рассмотренной программы — раскрытие аббревиатур в произвольном редакторе и контроль правильности написания слов. Раскрытие аббревиатуры происходит по внутренней таблице строк TSR-программы. Если набрана последовательность букв, после которой нажата “горячая” клавиша, обработчик прерывания 16h читает аббревиатуру, “стирает ее с экрана и в буфер клавиатуры или непосредственно в программу передает последовательность символов раскрытой аббревиатуры. Самый простой вариант этой программы — клавиатурные макросы. Это — передача в программу при нажатии какой-либо “горячей” клавиши целой строки символов.

3. УПРАВЛЕНИЕ АДАПТЕРОМ АСИНХРОННОЙ ПОСЛЕДОВАТЕЛЬНОЙ СВЯЗИ

3.1. Аппаратные средства асинхронной последовательной связи. Интерфейс RS-232

3.1.1. Формат слов информации

Большое число периферийных устройств персонального компьютера подключается к адаптеру асинхронной последовательной связи. Перенос информации между адаптером и внешним устройством организуется по правилам последовательного асинхронного интерфейса RS-232. Стандарт RS-232 регламентирует передачу последовательных двоичных потоков информации между интерфейсами или терминалами и связным оборудованием. Наиболее часто *адаптер асинхронной последовательной связи* используется для организации обмена информацией между компьютерами с использованием телефонных линий связи. В этом случае внешним устройством является специальный аппаратный узел — *модем* (образован из слов МОдулятор-ДЕМодулятор). Модулятор на передающей стороне преобразует последовательность битов, представляющих собой перепады уровня напряжения (цифровой сигнал), поступающие из адаптера последовательной связи, в аналоговые сигналы акустической частоты (логической единице соответствует сигнал с одной частотой, логическому нулю — с другой). Аналоговые сигналы передаются по телефонному проводу или преобразуются в радиосигнал

(так называемый *радиомодем*). На приемной стороне демодулятор преобразует аналоговый сигнал в цифровой код, который передается на вход адаптера.

Модем — достаточно сложное техническое устройство, способное “набирать” номер телефона, заданный компьютером, повторять автоматически эту процедуру через заданные промежутки времени, буферизовать информацию, распознавать передаваемые от компьютера команды, реализовывать функции автоответчика и др.

Информация по линиям интерфейса RS-232 передается асинхронно последовательным кодом. Это означает, что передатчик посылает байт данных бит за битом. Для такой последовательной передачи требуется только две линии (два провода). При передаче слов информации реализуется так называемый *старт-стопный метод*. Его суть в том, что каждое передаваемое слово начинается *старт-битом*, позволяющим приемнику определить начало передачи слова. Затем передается бит за битом байт информации. Завершение передачи слова отмечается специальными *стоп-битами*. Электрически логическому нулю в интерфейсе соответствует высокий потенциал +12 В, а логической единице — 0 В. Старт-бит — это всегда единица, стоп-бит (биты) — всегда нуль (нули). Таким образом, переключение напряжения на линии данных с +12 В на 0 В рассматривается приемной стороной как сигнал начала слова. По этому сигналу на приемной стороне запускается в работу специальный аппаратный узел — *сдвиговый регистр*, который “собирает” в параллельный код принятое бит за битом слово информации. Биты передаются с известной приемнику и передатчику частотой, измеряемой в битах в секунду (bps — bits per second). Передатчик и приемник используют разные источники синхронизации, которые работают с близкой, но все-таки различающейся частотой. Сильное расхождение частот приемника и передатчика вызывает возникновение специфической для асинхронной связи ошибки, называемой *ошибкой кадрирования (framing error)*.

Часто bps путают с единицей измерения бод. Иногда даже можно встретить такую единицу как бод в секунду.

Бод измеряет число сигналов в секунду, т. е. ссылка на единицу времени “встроена” в термин “бод”. Если один сигнал соответствует одному биту, то частота в бодах и в бит/с имеет одинаковое значение. Однако в некоторых интерфейсах связи может использоваться несколько потоков сигналов для передачи последовательности битов. Например, в стандарте Bell 212A для двухфазной передачи со скоростью 1200 bps используется два потока сигналов разной акустической частоты, каждый из которых работает на 600 бодах.

Сводная характеристика передаваемого в интерфейс слова называется *форматом слова*. Передача информации между источником и приемником возможна тогда, когда они используют одинаковый формат слова. Только в этом случае приемник может обнаружить конец слова. Если приемник полагает, что наступило время принимать стоп-биты (высокий потенциал), но из интерфейса поступает логическая единица, приемник фиксирует ошибку кадрирования.

Формат слова определяет следующие особенности переноса информации через интерфейс:

- 1) число битов, используемых для кодирования самого переносимого символа;
- 2) наличие или отсутствие контроля по четности;
- 3) способ формирования контрольного бита;
- 4) число стоп-битов.

3.1.2. Линии интерфейса RS-232

Полный стандарт RS-232C включает 25 линий, но на практике используется лишь некоторое их подмножество (табл. 3.1).

Обмен сигналами между адаптером компьютера и модемом строится по стандартизованному сценарию, в котором каждый сигнал генерируется сторонами лишь после наступления определенных условий или, говоря образно, “одна сторона ожидает реплики партнера и только потом произносит следующее слово”. Такая процедура обмена информацией называется *запрос-ответным режимом*, или

Табл. 3.1. Сигналы интерфейса RS-232

Номер линии	Обозначение	Использование сигнала
1		Экран
От компьютера в модем		
2	TXD	Передача данных в модем
4	RTS	Запрос на передачу (Request To Send): информирует модем о том, что компьютер хочет передать данные в линию
20	DTR	Готовность терминала данных (Data Terminal Ready): информирует о том, что в компьютере включено электропитание и он готов к обмену информацией
От модема в компьютер		
3	RXD	Прием данных от модема
5	CTS	Готовность модема (Clear To Send): информирует компьютер о том, что модем готов принимать данные от компьютера и передавать их в линию
6	DSR	Готовность набора данных (Data Set Ready): информирует компьютер о том, что модем включен в электросеть и готов к обмену данными
7	GND	Логический нуль
8	DCD	Соединение с модемом удаленной станции (Data Carrier Detected): информирует компьютер о том, что модем установил связь с модемом удаленной станции
22	RI	Индикатор вызова (Ring Indicator): информирует компьютер о том, что на телефон, к которому подключен модем, поступил "звонок"

“рукопожатием” (*handshaking*). Большинство из приведенных в табл. 3.1 сигналов как раз и нужны для аппаратной реализации “рукопожатия” между адаптером и модемом.

Обмен сигналами между сторонами интерфейса RS-232 выглядит так:

1) компьютер после включения питания выставляет сигнал DTR, который постоянно удерживается активным. Если модем включен в электросеть и исправен, он отвечает компьютеру сигналом DSR. Этот сигнал служит

подтверждением того, что DTR принят, и информирует компьютер о готовности модема к приему информации;

2) если компьютер получил сигнал DSR и хочет передать данные, он выставляет сигнал RTS;

3) если модем готов принимать данные, он отвечает сигналом CTS. Он служит для компьютера подтверждением того, что RTS получен модемом и модем готов принять данные от компьютера. С этого момента адаптер компьютера может бит за битом передавать информацию по линии TXD;

4) получив байт данных, модем может сбросить свой сигнал CTS, информируя компьютер о необходимости “притормозить” передачу следующего байта, например из-за переполнения внутреннего буфера;

5) программа компьютера, обнаружив сброс CTS, прекращает передачу данных, ожидая повторного появления CTS.

Когда модему необходимо передать данные в компьютер, он (модем) выставляет сигнал на разъеме 8 — DCD. Программа компьютера, принимающая данные, обнаружив этот сигнал, читает приемный регистр, в который сдвиговый регистр “собрал” биты, принятые по линии приема данных RXD. Когда для связи используются только приведенные в табл. 3.1 данные, компьютер не может попросить модем “повременить” с передачей следующего байта. Как следствие, существует опасность перепреопределения помещенного ранее в приемном регистре байта данных вновь “собранным” байтом. Поэтому при приеме информации компьютер должен очень быстро освобождать приемный регистр адаптера. В полном наборе сигналов RS-232 есть линии, которые могут аппаратно “приостановить” модем.

Компьютер может передать в модем множество команд (так называемых *управляющих строк*). Они опознаются модемом по символу (символам) начала управляющей строки. Наиболее часто таким символом служит ESC (ASCII 27). Управляющая строка может указывать номер телефона, который следует набрать для установления связи, инициализировать аппаратуру модема, послать уда-

ленной станции некоторый управляющий сигнал, выбрать частоту передачи и многое другое.

Обычно прием и передача информации выполняются одновременно, что с использованием современных модемов позволяет переносить информацию сразу в двух направлениях. Довольно часто возникает необходимость временной приостановки переноса данных. Для этого используется передача управляющих символов в потоке данных. Многие драйверы асинхронной связи построены по так называемому XON/XOFF протоколу. Прием любого символа XOFF (ASCII 11h) останавливает передачу данных из компьютера до момента получения сигнала XON (ASCII 13h). Естественно, использование XOFF и XON возможно только при передаче текстовых файлов.

Другая возможность управления темпом приема-передачи — обнаружение условия BREAK. Условие BREAK — это отсутствие в течение заранее оговоренного времени данных от передатчика. Обнаружив условие BREAK, приемник “уточняет” причину, передавая в передатчик согласованную заранее командную информацию.

3.1.3. Порты адаптера последовательной связи

Управление переносом информации по интерфейсу RS-232 выполняет специальная электронная схема, объединяющая до четырех адаптеров, каждый из которых обслуживает одну линию связи. MS-DOS ссылается на эти линии как на специальные символьные файлы с именами COM1 — COM4 (подробнее о специальных файлах см. в 2.1 второй книги комплекса). Синонимом COM1 является символьный файл AUX, связанный с первым адаптером последовательной связи. Физически компьютер может иметь и меньшее количество адаптеров последовательной связи. Определение числа адаптеров выполняется так же, как и для адаптеров параллельной связи либо по слову данных BIOSa, либо по слову, возвращаемому прерыванием 11h (подробнее об этом см. в 5.2).

Адаптер последовательной связи способен как передавать, так и принимать информацию из интерфейса. Осо-

бенностью адаптера является способность генерировать аппаратные прерывания по программируемой маске условий. IBM PC поддерживает аппаратно генерацию прерываний для последовательных адаптеров COM1 и COM2. Адаптер последовательной связи COM1 в компьютере IBM PC AT генерирует прерывание 0Ch (0Bh в компьютере IBM PC XT), адаптер COM2 — прерывание 0Bh (0Ch в компьютере IBM PC XT).

Каждый адаптер управляется десятью внутренними регистрами, доступ к которым осуществляется по номерам портов. Номера портов отсчитываются от *базового адреса (БА)*. Базовые адреса для каждого из адаптеров различны и хранятся в области данных по адресу 40:00h для COM1, 40:02h — для COM2, 40:04h — для COM3 и 40:06h — для COM4. Если по перечисленным адресам записан 0, это свидетельствует об отсутствии адаптера. Одни и те же адреса портов могут соответствовать различным внутренним регистрам в зависимости от направления передачи информации и значения бита 7 регистра управления линией. Внутренние регистры адаптера последовательной связи приведены в табл. 3.2.

Далее подробно описываются внутренние регистры адаптера, не используемые при управляемых прерываниями переносах информации.

Регистры данных (БА+0)

Использование регистра данных БА+0 зависит от состояния регистра управления линией (БА+3) и выполняемой операции доступа к порту (чтение или запись). Если адаптер принимает или передает информацию, в бите 7 регистра БА+3 должен быть записан 0. В этом случае запись байта в порт с номером БА+0 помещает данные в регистр данных передатчика. Отсюда они без каких-либо дополнительных управляющих воздействий передаются в сдвиговый регистр, который формирует слово информации и бит за битом выдает его в интерфейс. Чтение байта из внутреннего регистра БА+0 возвращает последний принятый адаптером байт данных.

Табл. 3.2. Внутренние регистры адаптера последовательной связи

Номер порта	Описание
БА + 0	При передаче или приеме данных (бит 7 регистра управления линией равен 0): регистр данных передатчика при записи в порт; регистр данных приемника при чтении из порта При задании формата слова адаптера (бит 7 регистра управления линией равен 1): делитель частоты (младший байт); как правило, только запись
БА + 1	При передаче или приеме данных (бит 7 регистра управления линией равен 0): регистр разрешения прерываний; как правило, только запись При задании формата слова адаптера (бит 7 регистра управления линией равен 1): делитель частоты (старший байт); как правило, только запись
БА + 2	Регистр идентификации прерывания; как правило, только чтение
БА + 3	Регистр управления линией; только запись
БА + 4	Регистр управления модемом; чтение и запись
БА + 5	Регистр состояния линии; только чтение
БА + 6	Регистр состояния модема; только чтение

Регистры делителя частоты (БА+0 и БА+1)

Если бит 7 регистра управления линией (БА+3) установлен в 1, содержимое регистра БА+3 определяет формат данных интерфейса. В этом случае порты с номерами БА+0 и БА+1 используются как защелки для так называемого *делителя частоты передачи(приема)*. Значение делителя — это число, полученное делением “магического” числа 1 843 200 на частоту передачи в бодах, умноженную на 16. “Магическое” число — это частота внутреннего источника синхронизации в 1 843 200 Гц стандартной микросхемы 8250 адаптера последовательной связи. Делитель частоты, равный 1, соответствует макси-

мально возможной частоте передачи в 115 200 бит/с. Это следует из того, что $1\,843\,200 / (115\,200 * 16) = 1$. Частоте передачи в 57 600 бит/с соответствует делитель частоты 2, частоте в 38 400 бит/с — делитель 3 и т.д.

Внутренний регистр управления линией (БА+3)

Используется для задания формата слова, передаваемого по интерфейсу, и некоторых дополнительных особенностей передачи или приема данных. Назначение его битов приведено в табл. 3.3.

Табл. 3.3. Биты регистра управления линией

Бит	Описание
7	Выбор содержимого в портах БА+0 и БА+1: = 0 - порт БА+0 используется для доступа к регистрам данных, а порт БА+1 - для доступа к регистру разрешения прерываний; = 1 - порты БА+0 и БА+1 используются для доступа к делителю частоты;
6	= 0 - обычное функционирование адаптера; = 1 - вызывает посылку в интерфейс сигнала BREAK
5, 4, 3	Выбор способа контроля по четности (паритета): = xx0 - отсутствие бита контроля по четности; = 001 - бит контроля формируется по четному паритету; = 011 - бит контроля формируется по нечетному паритету; = 101 - бит контроля равен 1; = 111 - бит контроля равен 0
2	Число стоп-битов: = 0 - 1 стоп-бит; = 1 - 2 стоп-бита
1, 0	Длина слова: = 00 - 5 бит; = 01 - 6 бит; = 10 - 7 бит; = 11 - 8 бит

Приведем пример Си-функции установки частоты передачи и формата слова, иллюстрирующей использование внутренних регистров БА+0, БА+1 и БА+3. Ее достоинством является возможность задания максимально допу-

стимых частот передачи символов, не поддерживаемых функцией BIOSa инициализации адаптера (см. 3.2).

```
/* L3_1.C
```

```
    Инициализация адаптера последовательной связи  
    непосредственным доступом к внутренним регистрам.  
    Устанавливается частота передачи baud_rate и формат  
    слова format. Значения битов полей format  
    соответствуют табл. 3.3. Значение baud_rate выбирается  
    из ряда 50, 75, 110, 134.5, 150, 300, 600, 1200, 1800,  
    2000, 2400, 4800, 9600, 19200, 38400, 57600, 38400,  
    115200.
```

```
    В н е ш н и е   п е р е м е н н ы е:
```

```
    unsigned base_addr – базовый адрес адаптера. */
```

```
#include <dos.h>
```

```
void set_baud(long unsigned baud_rate, char format)  
{
```

```
    extern unsigned base_addr;
```

```
    union {
```

```
        int baud_divisor;
```

```
        char bytes[2];
```

```
    } word;
```

```
    word.bytes[0] = 0;
```

```
    /* Вычисление значения частоты передачи  
    для записи в БА+0 и БА+1. */
```

```
    word.baud_divisor = 1843200L/ (long) baud_rate >> 4;
```

```
    /* Установка формата слова в БА+3. */
```

```
    outportb(base_addr + 3, format | 0x80);
```

```
    outportb(base_addr, word.bytes[0]);
```

```
    outportb(base_addr + 1, word.bytes[1]);
```

```
    /* Сброс бита 7 в регистре формата слова. */
```

```
    outportb(base_addr + 3, format);
```

```
}
```

Не следует при использовании телефонных линий задавать очень высокие (более 9600) частоты передачи. При организации связи двух компьютеров друг с другом без использования модемов (см. 3.4) возможно использование частот и более 19 200 бит/с.

Внутренний регистр состояния модема (БА+6)

Опрашивая регистр, компьютер отслеживает все изменения состояния модема, что позволяет определить возможность передачи байта (наличие сигнала CTS), необходимость приема информации (наличие сигнала DCD) и т.п. Четыре старших бита регистра показывают уровень напряжения на соответствующей линии интерфейса RS-232 (см. табл. 3.1), а четыре оставшихся — наличие изменения уровня напряжения с момента последнего чтения содержимого, так как любое чтение регистра обнуляет эти биты. Использование отдельных битов регистра приведено в табл. 3.4.

Табл. 3.4. Биты регистра состояния модема

Бит	Описание (если бит равен единице)
7	Сигнал на линии DCD
6	Сигнал на линии RI
5	Сигнал на линии DSR
4	Сигнал на линии CTS
3	Изменение на линии DCD
2	Изменение на линии RI
1	Изменение на линии DSR
0	Изменение на линии CTS

Внутренний регистр состояния линии (БА+5)

Этот регистр сообщает компьютеру информацию о состоянии как модема, так и тракта приема(передачи) адаптера (табл. 3.5).

Существуют два метода организации переноса данных по интерфейсу RS-232: последовательный опрос и управляемый прерываниями перенос. В самых простейших коммуникациях можно обойтись без прерываний от адаптера последовательной связи. В этом случае процедуру передачи данных по интерфейсу RS-232 можно представить следующим псевдокодом на языке Си:

Табл. 3.5. Биты регистра состояния линии

Бит	Описание (если бит равен единице)
7	Всегда 0
6	Сдвиговый регистр передатчика пуст
5	Буферный регистр передатчика пуст
4	Принят BREAK
3	Ошибка кадрирования (рассинхронизация передатчика и приемника)
2	Ошибка паритета
1	Ошибка переопределения данных
0	Готовность принятых данных

Передача символов

```

...
DO { /* Ожидание момента передачи символа. */
    WHILE(Бит 4 регистра БА+6
        "Сигнал на линии CTS" равен 0);
    /* Запись символа в регистр данных БА+0. */
    SEND(SYMBOL);
}
WHILE("Есть символы для посылки");
...

```

Прием символов

```

...
DO { /* Ожидание момента чтения данных. */
    WHILE(Бит 0 регистра БА+5
        "Готовность принятых данных" равен 0);
    /*Чтение символа из регистра данных БА+0. */
    READ(SYMBOL);
}
WHILE("Прием не остановлен");
...

```

Компьютер постоянно опрашивает порты данных и состояния для определения момента чтения принятого байта либо записи в порт данных очередного передаваемого байта. Постоянный опрос необходим вследствие того, что как при приеме, так и при передаче буферизуется только один байт данных и возможно его переопределение вновь поступающими данными.

Однако бесконечный цикл опроса превращает компьютер лишь в простейшую коммуникационную станцию, не способную выполнять какие-либо другие действия. Более предпочтительным является метод организации асинхронной связи, управляемой прерываниями. Технически адаптер последовательной связи позволяет генерировать прерывания при программируемых условиях. В случае управляемой прерываниями асинхронной связи используются описанные ниже внутренние регистры.

Внутренний регистр разрешения прерываний (БА+1)

Аппаратура адаптера последовательной связи может генерировать запросы аппаратных прерываний при возникновении в адаптере одной или нескольких ситуаций четырех возможных типов (табл. 3.6).

Табл. 3.6. Биты регистра разрешения прерываний

Бит	Описание (если бит равен единице)
7 - 4	Не используются
3	Разрешена генерация запроса прерывания при изменении значения регистра состояния модема (БА+6)
2	Разрешена генерация запроса прерывания при изменении значения регистра состояния линии (БА+5)
1	Разрешена генерация запроса прерывания в случае, когда пуст регистр передачи данных
0	Разрешена генерация запроса прерывания в случае готовности принятых данных

Запись в порт с номером БА+1 байта 00 запрещает генерацию запроса прерывания для всех возможных ситуаций. Запись значения 0Fh напротив разрешает генерацию запроса прерывания для всех типов ситуаций. Закрепление адаптеров последовательной связи за прерываниями, генерируемыми процессором, зависит от архитектуры персонального компьютера и подробно рассматривается в 1.3 второй книги комплекса. ISR прерываний от COM1 или COM2 может точно определить причину прерывания чтением регистра идентификации прерывания.

Внутренний регистр идентификации прерываний (БА+2)

Это — только читаемый внутренний регистр, позволяющий программе-обработчику прерывания определить причину, по которой сгенерировано прерывание или, другими словами, причину, по которой программа получила управление. Использование битов регистра приведено в табл. 3.7.

Табл. 3.7. Биты регистра идентификации прерываний

Бит	Описание
7 - 3	Не используются
2 - 1	Идентификатор причины прерывания: - 00 - прерывания по причине изменения регистра состояния модема (БА+6); - 01 - пуст регистр данных передатчика; - 10 - готовность данных в приемном регистре; - 11 - прерывание по причине изменения регистра состояния линии: либо установка бита ошибки, либо выделено условие BREAK
0	- 0 - прерывание не обработано (ожидающее обработки прерывание); - 1 - нет активных прерываний

Нули в битах 2 — 0 свидетельствуют о наличии еще не обслуженного прерывания, возникшего по причине изменения в линиях интерфейса RS-232, например из-за появления сигнала DSR. Устранение причины соответствующего прерывания выполняется чтением или записью соответствующего случаю регистра. Например, чтение регистра состояния модема “сбрасывает” условие прерывания и, если не происходит нового изменения состояния линий, прерывание по причине изменения состояния модема не генерируется. Аналогично запись байта в регистр данных передатчика “сбрасывает” прерывание из-за того, что этот регистр пуст. Чтение внутреннего регистра данных приемника “сбрасывает” прерывание по причине готовности данных.

Возможно одновременное возникновение причин для разных типов прерываний, и в этом случае аппаратура адаптера, выполняющая генерацию прерываний, упорядочивает их по приоритету. Прерывание в связи с изменением в регистре состояния модема всегда будет иметь

наивысший приоритет. Следующим по приоритету является прерывание по причине готовности принятых данных. После него генерируется прерывание из-за опустевшего регистра данных передатчика. И, наконец, наименьший приоритет имеет прерывание по причине изменения состояния линии.

Одному прерыванию 0Vh или 0Ch может соответствовать несколько причин сразу и устранение какой-то одной причины тем не менее не вызовет установки в 1 бита 0 регистра идентификации прерывания (БА+2). Поэтому обработчик прерывания не должен завершать свою работу до тех пор, пока не устранит все одновременно возникшие причины прерывания.

Внутренний регистр управления модемом (БА+4)

Строго говоря, этот регистр управляет работой не модема, а адаптера последовательной связи. Использование его битов приведено в табл. 3.8.

Табл. 3.8. Биты регистра управления модемом

Бит	Описание (если бит равен единице)
7 - 5	Не используются
4	Запуск автономного теста адаптера последовательной связи (LOOP)
3	Разрешает прерывания от адаптера (OUT2)
2	Определяемый пользователем запрос прерывания (OUT1)
1	Устанавливает сигнал на линии интерфейса RTS
0	Устанавливает сигнал на линии интерфейса DTR

3.2. Средства BIOSa управления асинхронной связью

Область данных BIOSa содержит информацию о числе адаптеров последовательной связи и об их базовых адресах. Определение числа адаптеров выполняется так же, как и для адаптеров параллельной связи, либо по слову данных BIOSa (слово по адресу 40:10h), либо по слову,

возвращаемому прерыванием 11h (подробнее об этом см. в 5.2). Далее приводится пример Си-функции, возвращающей число адаптеров последовательной связи, установленных в компьютере.

```
/* L3_2.C
   Возвращает число адаптеров
   последовательной связи компьютера. */

int com_inst(void)
{
    register unsigned _es * com = (unsigned _es *) 0x10;
    _ES = 0x40;
    return((*com & 0x0E00) >> 9); /* возвращает биты
                                   11, 10 и 9 слова по адресу 40:10h */
}
```

Базовые адреса адаптеров последовательной связи хранятся в области данных BIOSa по следующим адресам: 40:00h — для COM1; 40:02h — для COM2; 40:04h — для COM3; 40:06h — для COM4. Если слово равно нулю, это означает отсутствие соответствующего адаптера в системе.

BIOS предоставляет пользователю некоторую поддержку асинхронной последовательной связи. Управление адаптером и услуги по переносу данных выполняют 6 функций прерывания 14h. Для всех этих функций значение в регистре DX задает номер адаптера последовательной связи (0 для COM1, 1 для COM2 и т.д.). Все функции в АН возвращают копию внутреннего регистра состояния линии (см. табл. 3.5), но дополнительно бит 7 является *битом тайм-аута*.

Операции BIOSa с адаптером последовательной связи защищаются по тайм-ауту. Это означает, что по истечении определенного интервала времени и при отсутствии возможности выполнить функцию, функция BIOSa возвращает управление в точку вызова, сообщая ошибку тайм-аута. В результате не происходит “повисания” компьютера из-за ожидания неисполнимого в данный момент условия. Проблема состоит в том, что не все версии BIOSa поддерживают контроль по тайм-ауту.

Табл. 3.9. Формат слова информации при инициализации адаптера последовательной связи функцией АН = 00 прерывания 14h BIOS

Бит	Описание
7 - 5	Частота передачи: - 000 - 110 бит/с; - 001 - 150 бит/с; - 010 - 300 бит/с; - 011 - 600 бит/с; - 100 - 1200 бит/с; - 101 - 2400 бит/с; - 110 - 4800 бит/с; - 111 - 9600 бит/с
4, 3	Контроль по четности (паритет): - 00 или 10 - отсутствие контроля; - 01 - нечетный паритет; - 11 - четный паритет
2	Число стоп-битов: - 0 - один стоп-бит; - 1 - два стоп-бита
1, 0	Длина слова информации: - 00 - 5 бит; - 01 - 6 бит; - 10 - 7 бит; - 11 - 8 бит

Функция АН = 00 — задание формата слова данных, используемых адаптером для переноса (приема и передачи) данных. Значение в регистре AL задает параметры инициализации (табл. 3.9).

Функция в дополнение к обычному возврату состояния линии в АН в регистре AL возвращает копию внутреннего регистра состояния модема (см. табл. 3.4).

Версии BIOSa, соответствующие PS/2 (такой BIOS часто устанавливается и на IBM PC AT на базе микропроцессора 80386), поддерживают дополнительно функцию расширенной инициализации (функция АН = 04h), позволяющую устанавливать частоту передачи до 19 200 бит/с и длину слова информации от 5 до 8 бит. Инициализация адаптера последовательной связи может быть

выполнена независимо от наличия или отсутствия электрической связи с модемом и никогда не возвращает условие тайм-аута.

Приводимый далее фрагмент программного кода выполняет инициализацию адаптера следующими значениями: частота передачи 9 600 бит/с; 8 бит данных; отсутствие контроля четности; 1 стоп-бит.

```
/* Инициализация адаптера: 9 600 бит/с, 8 бит, 1
стоп-бит, отсутствие контроля по четности. */
struct REGPACK r;
...
r.r_ax = 0x00e3; /* было 00ff */
r.r_dx = com_port - 1;
intr(0x14, &r); /* функция AH = 00
                  прерывания 14h BIOSa */
...
```

Функция AH = 01h используется для записи символа из регистра AL в регистр передатчика адаптера. По возвращении значение AL не изменяется, а AH сообщает состояние линии. Если BIOS не поддерживает контроль по тайм-ауту, вызов этой функции приводит к бесконечному ожиданию при отсутствии связи с приемником.

Функция AH = 02h используется для чтения символа в регистр AL из регистра данных приемника адаптера. По возвращении в AL содержится прочитанный байт данных, а AH сообщает состояние линии.

Функция AH = 03h используется для определения состояния адаптера последовательной связи. В дополнение к обычному возврату состояния линии в AH, в регистре AL возвращается копия внутреннего регистра состояния модема (см. табл. 3.4).

Версии BIOSa, соответствующие PS/2, поддерживают дополнительно функцию AH = 05h, выполняющую расширенное управление адаптером. Далее описываются две подфункции, выбираемые регистром AL.

Подфункция AL = 00 возвращает значение внутреннего регистра управления модемом (БА+4). В регистре AX возвращается состояние модема и линии, а в регистре

BL сообщается текущее значение внутреннего регистра управления модемом (см. табл. 3.8).

Подфункция AL = 01 устанавливает внутренний регистр управления модемом в значение, задаваемое регистром BL.

Turbo C имеет библиотечную функцию bioscom(), выполняющую действия, аналогичные функциям BIOSa AH = 00 — 02.

Недостатком функций BIOSa доступа к адаптеру последовательной связи является то, что они правильно работают только при наличии сигналов минимум на 9 линиях, приведенных в табл. 3.1 (подробнее об этом см. в 3.4).

3.3. Управляемая прерываниями асинхронная связь

3.3.1. Настройка адаптера на генерацию прерываний

Как отмечалось ранее, существуют две стратегии организации обмена данными по интерфейсу RS-232:

- 1) управляемый прерываниями обмен данными;
- 2) последовательный опрос.

В первом случае адаптер последовательной связи инициализируется так, что те или иные события в адаптере и линиях интерфейса генерируют аппаратные прерывания. Эти прерывания обслуживает программа-обработчик, которая принимает символы из интерфейса и помещает их в приемный буфер либо передает очередной байт в интерфейс. Другие программы читают информацию уже из буфера в памяти или записывают ее в буфер при необходимости передать блок информации другому компьютеру. Операция записи информации в буфер активизирует асинхронную передачу.

Во втором случае компьютер выполняет бесконечный цикл опроса внутренних регистров адаптера. При передаче он ожидает момента возникновения сигнала от мо-

дема CTS, а при приеме — момента наступления события “Готовность байта данных”. Программное обеспечение для обмена в этом случае намного проще, но отсутствует возможность выполнения других программ. Дело в том, что момент “Готовность байта данных” нельзя оставить в ожидании обслуживания (чтения регистра данных), так как принятый байт может быть переопределен следующим байтом. Аппаратно чтение регистра состояния линии или непосредственным доступом к порту, или через BIOS “сбрасывает” состояние “Готовность байта данных”, и адаптер принимает в приемный регистр очередной байт.

Управляемый прерываниями обмен может объединяться с последовательным опросом. Например, передача символов в интерфейс может выполняться программой без использования аппаратных прерываний, а прием информации — осуществляться по прерываниям. В данном параграфе рассматриваются:

- 1) инициализация адаптера последовательной связи для организации обмена данными по прерываниям;

- 2) структура обработчика прерывания адаптера асинхронной связи, рассчитанного на асинхронную передачу и асинхронный прием;

- 3) функции записи информации в буфер обработчика прерываний с активизацией асинхронной передачи;

- 4) функции чтения информации из буфера обработчика прерываний с активизацией асинхронного приема.

Для организации передачи информации, управляемой прерываниями, необходимо прежде всего инициализировать адаптер так, чтобы он генерировал запросы аппаратных прерываний, установить собственный обработчик прерывания и изменить маску запросов аппаратных прерываний в контроллере 8259. Инициализация адаптера состоит из следующих шагов:

- 1) маскируются аппаратные прерывания, так как вся процедура инициализации адаптера должна выполняться без прерываний;

- 2) сохраняется предыдущий вектор прерывания: 0Ch для COM1, 0Bh для COM2 в компьютере IBM PC AT; 0Bh для COM1, 0Ch для COM2 в компьютерах IBM PC

и IBM PC XT (подробнее о закреплении векторов прерываний и внешних устройств см. во второй книге комплекса);

3) на место сохраненного вектора записывается точка входа в собственный обработчик прерывания;

4) в контроллере аппаратных прерываний деблокируются прерывания от COM1 или COM2;

5) в адаптере последовательной связи деблокируются прерывания. Для этого маска разрешаемых прерываний записывается во внутренний регистр разрешения прерываний (БА+1);

6) разрешаются прерывания во внутреннем регистре управления модемом (БА+4);

7) очищаются все условия прерываний, которые, возможно, уже возникли из-за включения механизма прерываний в работу;

8) вновь разрешаются аппаратные прерывания.

Приведенная последовательность состоит из минимально возможного числа шагов и при необходимости может дополняться сохранением состояния внутренних регистров адаптера последовательной связи и контроллера прерываний, установкой частоты передачи и формата слова, проверкой наличия связи с удаленным компьютером и т.д. Кратко прокомментируем выполняемые шаги.

Маскирование прерываний перед началом инициализации позволяет избежать ситуации, при которой процесс инициализации будет прерван аппаратным прерыванием, в том числе и от адаптера последовательной связи, на “полпути”, когда не закончено, например, переключение векторов прерываний. Наиболее просто маскирование прерываний в программе на Си можно выполнить, используя макро `disable()`. После завершения критической секции кода программы следует вновь разрешить прерывания от аппаратуры, используя макро `enable()`.

Установки маски в регистр разрешения прерываний (БА+1) недостаточно для того, чтобы адаптер последовательной связи генерировал запросы прерывания. Необходимо также установить в 1 бит 3 внутреннего регистра управления модемом (БА+4), что, между прочим, часто

забывают сделать в тех немногих примерах, которые можно найти в литературе.

Программирование адаптера влияет только на генерацию им запросов прерываний по линиям IRQ3 и IRQ4. Однако будут или не будут прерывания от адаптера последовательной связи обслуживаться центральным процессором (т.е. будут ли происходить прерывания), зависит от текущего состояния контроллера прерываний. Эта микросхема используется для задания номера прерывания, выполняемого процессором при возникновении сигналов на линиях запросов прерываний, для упорядочивания аппаратных прерываний по приоритетам и др. (Подробнее о работе контроллера прерываний см. в 1.3 второй книги комплекса.)

Контроллер прерываний обслуживает обычно до 16 линий запросов прерываний IRQ0 — IRQ15. Приоритет запроса убывает от IRQ0 (наивысший приоритет) к IRQ15 (наинизший приоритет). Закрепление узлов персонального компьютера за линиями запросов является одним из отличительных признаков архитектур персонального компьютера. В архитектуре IBM PC линии запросов прерываний закреплены за различными аппаратными узлами следующим образом (табл. 3.10).

Ту или иную линию запроса аппаратного прерывания можно маскировать. Маска воспринимаемых запросов хранится во внутреннем регистре контроллера прерываний 8259 и может быть прочитана или записана доступом к порту 21h. Каждый бит этого порта соответствует одной линии запроса прерываний. Если этот бит равен 1, линия маскирована, запросы прерываний “невидимы” контроллеру и, следовательно, не обслуживаются процессором. В случае, когда бит равен нулю, запрос прерывания будет обслуживаться в соответствии с приоритетом. Когда поступает запрос аппаратного прерывания, контроллер анализирует значение, записанное во внутреннем регистре обслуживания прерываний (порт 20h). В этом регистре хранится уровень прерывания, обслуживаемого в текущий момент. Если он ниже уровня приоритета нового запроса, начинается обслуживание этого запроса. Контроллер со-

Табл. 3.10. Соответствие линий запросов прерываний аппаратным узлам персонального компьютера

Линия запроса	Закрепление узла аппаратуры
IRQ0	Аппаратный таймер
IRQ1	Клавиатура
IRQ2	Канал ввода/вывода (сигнал от второй микросхемы 8259 в IBM PC AT)
IRQ3	1-й адаптер последовательной связи (COM1) (для IBM PC AT - COM2)
IRQ4	2-й адаптер последовательной связи (COM2) (для IBM PC AT - COM1)
IRQ5	Контроллер фиксированного диска
IRQ6	Контроллер накопителей на гибком диске
IRQ7	Контроллер 1-го адаптера параллельной связи (LPT1)

общает в процессор о необходимости выполнения прерывания и передает его номер. Процессор “генерирует” инструкцию INT с переданным номером. Закрепление номеров прерываний за линиями запроса может изменяться программированием контроллера. Обычно линии IRQ0 соответствует номер прерывания 8, линии IRQ1 — 9, IRQ2 — 0Ah, IRQ3 — 0Bh, IRQ4 — 0Ch и т.д. (Подробнее о работе контроллера прерываний см. в 1.3 второй книги комплекса.)

Важно, чтобы по завершении аппаратного прерывания его уровень приоритета был бы сброшен в порте 20h. В противном случае менее приоритетные запросы прерываний будут по-прежнему игнорироваться несмотря на то, что более приоритетное прерывание уже обслужено. Сброс внутреннего регистра обслуживания прерываний выполняется записью в порт 20h значения 20h.

Далее рассматривается пример функции `install_com()`, устанавливающей на вектор прерывания `com_interrupt` (0Bh или 0Ch) обработчик прерывания `com_handler` для заданного адаптера последовательной связи `com_adapter` (0 — COM1, 1 — COM2). Функция предназначена для совместного использования с функцией `uninstall_com()`, восстанавливающей аппаратные и программные средства персональ-

ного компьютера в то состояние, в котором они находились до установки собственного обработчика прерываний и настройки адаптера на генерацию аппаратных прерываний. Для упрощения процесса восстановления функция `install_com()` сохраняет во внешних структурных переменных по шаблонам `struct COM_STATE` и `struct ISR` соответственно состояние аппаратуры и предыдущий вектор прерывания. Последовательность выполняемых действий ясна из комментариев к тексту функции.

```
/* L3_3.C
```

```
    Инсталлирует обработчик прерывания com_handler аппаратного прерывания com_interrupt (0Bh или 0Ch) от адаптера последовательной связи com_adapter (0 – COM1, 1 – COM2). Настраивает адаптер на генерацию всех возможных типов запросов прерываний и деблокирует контроллер прерываний 8259. Рассчитана на архитектуру IBM PC AT.
```

```
    В о з в р а т:
```

```
    NO_COM (-1) – отсутствует запрошенный адаптер  
                  либо он не COM1 или COM2;
```

```
    NOT_READY (-2) – модем не отвечает на сигнал DTR  
                     сигналом DSR;
```

```
    NO_CTS (-3) – модем не отвечает сигналом CTS.
```

```
*/
```

```
#include <dos.h>
```

```
#define NO_COM      -1
```

```
#define OK          0
```

```
#define NOT_READY   -2
```

```
#define NO_CTS      -3
```

```
int com_inst(void);
```

```
struct ISR
```

```
{
```

```
    unsigned isr_offset; /* смещение точки входа  
                          обработчика прерываний */
```

```
    unsigned isr_segment; /* сегмент точки входа  
                           обработчика прерываний */
```

```
    unsigned isr_number; /* номер вектора прерываний */
```

```
};
```



```

struct COM_STATE
{
    char port_21; /* значение порта 21 контроллера 8259 */
    char mcr; /* значение регистра управления модемом */
    char ier; /* значение регистра разрешения прерываний */
    char lcr; /* значение регистра управления линией */
    char baud0; /* младший байт частоты передачи */
    char baud1; /* старший байт частоты передачи */
};

```

```

extern struct ISR old_isr;
extern struct COM_STATE old;
extern unsigned base_addr;

```

```

install_com(int com_interrupt,
            void interrupt (*com_handler)(),
            int com_adapter)
{
    struct REGPACK r;
    unsigned far * bios_addr=(unsigned far *) 0x00400000L;
    int i = 0xff; /* отработка задержки */
    /* Проверка наличия в системе запрошенного адаптера. */
    if((com_inst() < com_adapter) || (com_adapter > 1))
        return NO_COM;
    base_addr = * (bios_addr + com_adapter);
    if(!base_addr)
        return NO_COM;
    /* Сохранение "старого" вектора прерывания. */
    r.r_ax = (0x35 << 8) | com_interrupt;
    intr(0x21, &r);
    old_isr.isr_offset = r.r_bx;
    old_isr.isr_segment = r.r_es;
    old_isr.isr_number = com_interrupt;
    /* Сохранение предыдущего состояния аппаратных средств
        адаптера и контроллера прерываний 8259. */
    old.port_21 = inportb(0x21);
    old.ier = inportb(base_addr + 1);
    old.lcr = inportb(base_addr + 3);
    old.mcr = inportb(base_addr + 4);
    outportb(base_addr + 3, old.lcr | 0x80); /* установка
                                                бита 7 в 1 */
}

```



```

old.baud0 = inportb(base_addr);
old.baud1 = inportb(base_addr + 1);
outportb(base_addr + 3, old.lcr);
/* Установка обработчика прерывания com_handler. */
disable();
r.r_ax = (0x25 << 8) | com_interrupt; /* функция
                                         MS-DOS установки вектора прерывания */
r.r_dx = FP_OFF(com_handler);
r.r_ds = FP_SEG(com_handler);
intr(0x21, &r);
/* Демаскирование прерываний
   в контроллере 8259 (IBM PC AT). */
outportb(0x21, inportb(0x21) & ~ (0x10 >> com_adapter));
/* Программирование регистра разрешения прерываний:
   разрешены все возможные типы прерываний. */
outportb(base_addr + 1, 0x0F);
/*outportb(base_addr + 1, 0x01);*/ /* для примера L3_7.C
   следует использовать эту строку вместо предыдущей */
/* Программирование регистра управления модемом:
   установка сигнала на линии интерфейса DTR;
   обработка задержки и анализ ответа модема;
   установка сигналов на линиях интерфейса DTR и RTS;
   обработка задержки и анализ ответа модема;
   разрешение запросов прерывания от адаптера. */
outportb(base_addr + 4, old.mcr | 0x01); /*установка DTR*/
while(i--); /* см. примечания к L3_7.C */
if(!(inportb(base_addr + 6) & 0x20))
    return NOT_READY;
outportb(base_addr + 4, old.mcr | 0x03); /*установка DTR
                                         и RTS */
i = 0xff;
while(i--);
if(!(inportb(base_addr + 6) & 0x10))
    return NO_CTS; /* см. примечания к L3_7.C */
outportb(base_addr + 4, old.mcr | 0x0B);
/* "Сброс" причины возможных прерываний в адаптере. */
outportb(base_addr, 0x00); /* "пуст регистр сдвига
                             передатчика" */
inportb(base_addr); /* "готовность данных" */
inportb(base_addr + 6); /* "изменение регистра
                         состояния модема" */

```



```

    inportb(base_addr + 5); /* "изменение регистра
                               состояния линии" */
    enable(); /* вновь разрешаем аппаратные прерывания */
    return OK;
}

```

Приведенная функция настраивает адаптер на генерацию запросов аппаратных прерываний по всем возможным причинам, поэтому во внутренний регистр разрешения прерываний записывается маска, например, 0x0f. В случае, когда необходима активизация обработчика только при получении из интерфейса очередного байта данных, записывается другая маска прерываний. Внутренний регистр управления модемом (БА+4) используется не только для демаскирования прерываний, но и для начального этапа установления связи с модемом. Поэтому сначала устанавливается логическая единица на линии DTR. Этот сигнал удерживается в течение всего времени работы адаптера. При нормальной работе модема он должен отвечать на DTR сигналом на линии DSR. Для обнаружения сигнала на этой линии считывается внутренний регистр состояния модема (БА+6), в котором выделяется бит 5. Если бит 5 равен 1, диалог с модемом продолжается. Если бит 5 равен 0, это, скорее всего, свидетельствует об отсутствии связи с модемом или о том, что он отключен. Продолжением диалога является сигнал на линии RTS, ответом на который должен быть сигнал CTS модема. Наличие или отсутствие логической единицы на линии CTS обнаруживается чтением байта из регистра внутреннего состояния модема. В этом байте выделяется бит 4. В функции в качестве задержки используется "пустой" цикл на 0xFF проходов. Однако для сильно удаленного от компьютера модема или высокопроизводительного процессора этого времени может просто не хватить на получение ответного сигнала. В таком случае следует увеличить время ожидания ответного сигнала из интерфейса.

После того, как отпадет необходимость в использовании асинхронной связи, следует восстановить адаптер,

контроллер прерываний 8259 и таблицу векторов прерываний в исходное состояние. Эти действия выполняет приводимая далее функция `uninstall_com()`. Логика ее работы достаточно подробно освещена комментариями в тексте функции.

```
/* L3_4.C
```

```
    Восстанавливает аппаратуру адаптера, контроллер прерываний 8259 и таблицу векторов прерываний, модифицированные при инициализации прерываний функцией install_com(). Деинсталлирует обработчик прерывания com_handler аппаратного прерывания com_interrupt (0Bh или 0Ch) от адаптера последовательной связи com_adapter (0 – COM1, 1 – COM2). Рассчитана на архитектуру IBM PC AT.
```

```
    В о з в р а т:
```

```
    NO_COM (-1) – отсутствует запрошенный адаптер  
                  либо он не COM1 или COM2. */
```

```
#include <dos.h>
```

```
#define OK          0
```

```
#define NO_COM      -1
```

```
int com_inst(void);
```

```
struct ISR
```

```
{  
    unsigned isr_offset; /* смещение точки входа  
                           обработчика прерываний */  
    unsigned isr_segment; /* сегмент точки входа  
                           обработчика прерываний */  
    unsigned isr_number; /* номер вектора прерываний */  
};
```

```
struct COM_STATE
```

```
{  
    char port_21; /* значение порта 21h контроллера 8259 */  
    char mcr;     /* значение регистра управления модемом */  
    char ier;     /* значение регистра разрешения прерываний */  
    char lcr;     /* значение регистра управления линией */  
    char baud0;   /* младший байт частоты передачи */  
    char baud1;   /* старший байт частоты передачи */  
};
```



```

extern struct ISR old_isr;
extern struct COM_STATE old;
extern unsigned base_addr;

int uninstall_com(int com_adapter)
{
    struct REGPACK r;
    unsigned far * bios_addr=(unsigned far *) 0x00400000L;
    /* Проверка наличия в системе запрошенного адаптера. */
    if((com_inst() < com_adapter) || (com_adapter > 1))
        return NO_COM;
    base_addr = * (bios_addr + com_adapter);
    if(!base_addr)
        return NO_COM;
    disable(); /* маскирование аппаратных прерываний */
    /* Восстановление таблицы векторов прерываний. */
    r.r_ax = (0x25 << 8) | old_isr.isr_number;
    r.r_dx = old_isr.isr_offset;
    r.r_ds = old_isr.isr_segment;
    intr(0x21, &r);
    /* Восстановление состояния адаптера
       и контроллера 8259. */
    outportb(0x21, old.port_21);
    outportb(base_addr + 1, old.ier);
    outportb(base_addr + 3, old.lcr);
    outportb(base_addr + 4, old.mcr);
    outportb(base_addr + 3, inportb(base_addr + 3) | 0x80);
    outportb(base_addr, old.baud0);
    outportb(base_addr + 1, old.baud1);
    outportb(base_addr + 3, old.lcr);
    /* "Сброс" причин возможных прерываний в адаптере. */
    outportb(base_addr, 0x00); /* "пуст регистр сдвига
                                передатчика" */
    inportb(base_addr); /* "готовность данных" */
    inportb(base_addr + 6); /* "изменение регистра
                             состояния модема" */
    inportb(base_addr + 5); /* "изменение регистра
                             состояния линии" */
    enable(); /* разрешение аппаратных прерываний */
    return OK;
}

```


3.3.2. Структура обработчика прерываний асинхронной связи

Обработчик прерывания, в том числе и обработчик прерывания от адаптера последовательной связи, должен удовлетворять ряду общих требований, подробно рассмотренных в п.2.1.3. Кроме того, обработчик прерываний от адаптеров последовательной связи имеет свои специфические черты:

1) рассматриваемый обработчик является обработчиком аппаратного прерывания и поэтому должен завершаться инструкциями очистки внутреннего регистра обслуживаемого прерывания контроллера 8259 (порт 20h). Это выполняет следующий фрагмент Си-кода:

```
...  
outportb(0x20, 0x20);  
...
```

2) получив управление, обработчик уточняет причину возникновения прерывания. Для этого читается и анализируется содержимое внутреннего регистра идентификации прерывания (БА+2);

3) определив причину прерывания, обработчик устраняет ее и выполняет все необходимые действия. Например, если прерывание вызвано готовностью данных, обработчик считывает регистр данных приемника, что устраняет причину прерывания. Затем он помещает принятый байт в приемный буфер.

4) перед завершением своей работы обработчик прерывания обязательно анализирует наличие еще не обработанных прерываний. Как отмечалось ранее, различные причины прерываний могут возникать одновременно. Поэтому после “сброса” причины и выполнения всех необходимых действий вновь анализируется регистр идентификации прерывания. Если его бит 0 равен 0, это означает, что устранены все причины и обработчик может завершать свою работу.

Перечисленные действия обязательно должны выполняться в обработчике прерываний адаптера последовательной связи. Другие дополнительные действия обработчика связаны с используемым протоколом обмена информацией. Например, если используется протокол остановки(возобновления) передачи по принятым символам XOFF/XON, обработчик, во-первых, должен анализировать принимаемые символы и, во-вторых, получив символ XOFF, останавливать передачу до получения символа XON. Если протокол поддерживает условие BREAK, обработчик обнаруживает возникновение этого условия, анализируя внутренний регистр состояния линии. Выделив условие, обработчик выполняет заранее оговоренные действия.

В качестве простейшего примера обработчика прерываний адаптера последовательной связи далее приводится “отладочный” обработчик, позволяющий вывести на экран в шестнадцатеричной форме все байты данных, принимаемые компьютером от манипулятора “мышь”, подключаемого к адаптеру последовательной связи компьютера IBM PC AT:

```
/* L3_5.C
```

```
    Модель обработчика прерываний от адаптера
последовательной связи для приема символов от “мыши”.
Предварительно должен быть запущен драйвер,
обслуживающий “мышь”. При инициализации адаптера
разрешаются только прерывания по готовности данных.
Принимаемые байты выводятся непосредственно в
видеобуфер. */
#include <dos.h>
void byte_in_hex(char, char *); /* преобразование байта
                                в два символа, записанных в 16 с/с */
void interrupt com_handler()
{
    extern unsigned base_addr;
    extern char received_byte;
    static char far * vid_mem = (char far *)0xb8000000L;
    static char * ptr;
    static char string[5] = {0x00,0x00,0x20,0x20, 0};
    static int page = 0;
    static char atr = 0x7;
```



```

/* Чтение байта из регистра данных приемника. */
received_byte = inportb(base_addr); /* "заглушка" */
byte_in_hex(received_byte, string);
string[2] = 0x20;
/* Вывод на экран принятого байта. */
ptr = (char *) string;
while(*ptr)
{
    *vid_mem++ = *ptr++;
    *vid_mem++ = atr;
    page ++;
    if(page == 2000)
    {
        vid_mem = (char far *) 0xb8000000L;
        page = 0;
        atr ++;
    }
}
outportb(0x20, 0x20);
}

```

Вызываемая обработчиком функция `byte_in_hex()` преобразует получаемый байт в ASCIIZ-строку из двух символов, содержащую шестнадцатеричное представление принятого байта данных:

```

/* L3_6.C
Преобразует байт в два символа 16 с/с. */
void byte_in_hex(char byte, char * string)
{
    register char work;
    work = (byte & 0xf0) >> 4; /* старшая цифра
                                кода байта */
    if(work >= 0 && work <= 9)
        work += '0';
    else
        work += 0x37; /* 'A' - 0x0A */
    string[0] = work;
    work = byte & 0x0f; /* младшая шестнадцатеричная
                        цифра байта */
    if(work >= 0 && work <= 9)
        work += '0';
}

```



```

else
    work += 0x37;    /* 'A' - 0x0A */
    string[1] = work;
    string[2] = 0;  /* нуль-терминатор
                     строки из двух символов */
}

```

Скомпилировав совместно файлы L3_5.C, L3_6.C, L3_3.C, L3_4.C и приводимую далее функцию main() (L3_7.C), можно получить полезную утилиту, выводящую на экран все байты данных, передаваемые “мышью” или любым другим периферийным устройством в адаптер последовательной связи:

```

/* L3_7.C
   Отладочная программа инсталляции обработчика
   прерывания 0Ch; работает для IBM PC AT. */
#include <dos.h>
#include <bios.h>
#include <stdio.h>
/* Прототипы используемых функций. */
void interrupt com_handler();
int com_inst(void);
int install_com(int, void interrupt (), int);
int uninstall_com(int);
struct ISR
{
    isr_offset; /* смещение точки входа
                  обработчика прерываний */
    isr_segment; /* сегмент точки входа
                  обработчика прерываний */
    isr_number; /* номер вектора прерываний */
};
struct COM_STATE
{
    char port_21; /* значение порта 21 контроллера 8259 */
    char mcr;     /* значение регистра управления модемом */
    char ier;     /* значение регистра разрешения прерываний */
    char lcr;     /* значение регистра управления линией */
    char baud0;   /* младший байт частоты передачи */
    char baud1;   /* старший байт частоты передачи */
};

```



```

struct ISR old_isr;
struct COM_STATE old;
unsigned base_addr;
char received_byte;
void main(void)
{
    int ret;
    ret = install_com(0x0c, com_handler, 0);
    printf("Возврат из инсталляции %d\n", ret);
    printf("Вывод байтов, передаваемых \"мышью\". \"\n\n\
        \"Для завершения нажмите любую клавишу ...\"");
    bioskey(0); /* ожидание нажатия любой клавиши для
        завершения выполнения программы */
    /* Восстановление состояния адаптера. */
    uninstall_com(0);
}

```

Рассматриваемая программа имеет несколько особенностей. Перед ее использованием необходимо запустить драйвер, обслуживающий "мышь". Драйвер установит поддерживаемую "мышью" частоту передачи и формат слова. Так как данный пример рассчитан лишь на распечатку принимаемых от "мыши" байтов данных, во внутреннем регистре разрешения прерываний адаптера (БА+1) необходимо разрешить только прерывания по причине готовности данных, т. е. в регистр БА+1 во время инициализации записать маску 0x01, а не 0x0f, как это было сделано в приведенном примере функции install_com(). Кроме того, из листинга L3_3.C следует исключить выделенный фоном фрагмент. В этой части функция install_com() выполняет диалог с модемом. Сначала функция инициирует сигнал на линии DTR и ожидает ответного сигнала на линии DSR. При его получении функция инициирует сигнал RTS и ожидает ответного сигнала на линии CTS. "Мыши" при своем подключении к адаптеру используют только линии GND, RTS, DTR, TXD и RXD (см. табл. 3.1). Поэтому при работе с "мышью" функция install_com() никогда не дожидается ответа по линиям DSR и CTS и будет аварийно завершаться, сообщая условие NOT_READY.

Далее приводится общая модель обработчика прерываний от адаптера, работающего по прерываниям как при приеме, так и при передаче символов. Обработчик прерывания выполняет обнаружение следующих особых условий: прием символа XOFF и выделение BREAK. Реакция обработчика на эти условия зависит от принятого протокола.

```
/*
COM_ISR.C
Модель обработчика прерываний от адаптера
последовательной связи. Внешняя перемен-
ная - extern unsigned base_addr - базовый
адрес адаптера последовательной связи.
*/

#include <dos.h>

#define XOFF 0x11
void interrupt com_handler()
{
    static char result_isr, line_status, modem_status,
               received_byte;
    extern unsigned base_addr;
    /* . . . . . */
    /* Анализ причины прерывания чтением регистра
       идентификации прерывания. */
    result_isr = inportb(base_addr + 2);
    do
    {
        if(!(result_isr & 0x01)) /* есть ли активные
                                прерывания ? */
        {
            /* Выделение причины прерывания. */
            switch((result_isr & 0x06) >> 1) {
            case 0:
                /*изменение регистра состояния модема*/
                * Сброс причины прерывания и анализ
                  состояния линий модема. */

                modem_status = inportb(base_addr + 6);
                if(modem_status & 0x01)
```



```

        /* Секция обработки ситуации
        "Изменение сигнала на линии CTS". */;
else
    if(modem_status & 0x02)
        /* Секция обработки ситуации
        "Изменение сигнала на линии DSR". */;
    else
        if(modem_status & 0x04)
            /* Секция обработки ситуации
            "Изменение сигнала на линии RI". */;
        else
            if(modem_status & 0x08)
                /* Секция обработки ситуации
                "Изменение сигнала на линии DCD". */;
break;
case 1:
    /* пуст регистр данных передатчика */
    /* Извлечение символа из буфера
    передачи и запись в регистр
    данных передатчика. */
break;
case 2:
    /* готовность данных в приемном регистре */
    /* Чтение байта из регистра данных
    приемника и его запись
    в приемный буфер данных. */
    if((received_byte =
        inportb(base_addr)) == XOFF)
        /* Секция обработки ситуации
        получения XOFF. */;
    else
        /* Обработка принятого символа. */
break;
case 3:
    /*изменение регистра состояния линии*/
    /* Сброс причины прерывания и анализ
    состояния линии. */
    line_status = inportb(base_addr + 5);
    if(line_status & 0x01)

        /* Секция обработки ситуации
        "Готовность данных". Эта ситуация

```



```

        всегда возникает вместе с ситуацией
        "Прерывание по причине готовности
        данных", но имеет более низкий
        приоритет и поэтому обрабатывается
        после приема данных. */;
    else
        if(line_status & 0xE0)
            /* Секция обработки ситуации возник-
            новения одной из ошибок данных.*/;
        else
            if(line_status & 0x10)
                /* Секция обработки
                ситуации BREAK.*/;
            else
                if(line_status & 0x20)
                    /* Секция обработки ситуации
                    "Пуст буферный регистр данных
                    передатчика". Эта ситуация всегда
                    возникает вместе с ситуацией
                    "Прерывание из-за пустого регистра
                    данных передатчика данных", но
                    имеет более низкий приоритет и
                    обрабатывается после передачи
                    данных. */;
                else
                    if(line_status & 0x40)
                        /* Секция обработки ситуации
                        "Пуст сдвиговый регистр данных
                        передатчика". */;
                    break;
            }
        }
    else
        break; /* выход из цикла обработки
        различных прерываний */
}
while(1);
/* "Очистка" внутреннего регистра идентификации
обслуживаемого прерывания в контроллере 8259. */
outportb(0x20, 0x20);
}

```


3.4. Организация связи между компьютерами по интерфейсу RS-232

3.4.1. Схемы немодемного соединения компьютеров друг с другом

Большое число периферийных устройств подключаются к интерфейсу RS-232 без использования модема (так называемое *немодемное соединение*). В этом случае периферийное устройство “обманывает” адаптер, “притворяясь” модемом. Используя специально изготовленные соединительные кабели, можно организовать связь и между двумя компьютерами. Далее рассматриваются способы подключения немодемных устройств к адаптеру персонального компьютера и обсуждаются особенности построения коммуникационного программного обеспечения для каждого из них.

Адаптер асинхронной последовательной связи может иметь два типа разъемов для подключения периферийных устройств к адаптеру асинхронной последовательной связи: полный 25-контактный разъем и 9-контактный разъем. В табл. 3.11 приведено возможное соответствие контактов для 9- и 25-контактных разъемов.

Табл. 3.11. Соответствие контактов 25- и 9-контактных разъемов адаптеров асинхронной последовательной связи

Мнемоника сигнала	Номер контакта для разъема	
	25 контактов	9 контактов
DCD	8	1
RXD	3	2
TXD	2	3
DTR	20	4
GND	7	5
DSR	6	6
RTS	4	7
CTS	5	8
RI	22	9

Как видно из таблицы, 9-контактный разъем включает минимальный набор сигналов для организации двухсторонней связи между компьютером и периферийным устройством с обеспечением аппаратного запрос-ответного режима.

Минимальное число линий для организации двухсторонней связи компьютера с модемным устройством равно 3: для приема данных RXD, передачи данных TXD и для сигнального заземления GRD. Связь между компьютерами или компьютером и периферийным устройством в этом случае организуется по схеме, приведенной на рис. 3.1.

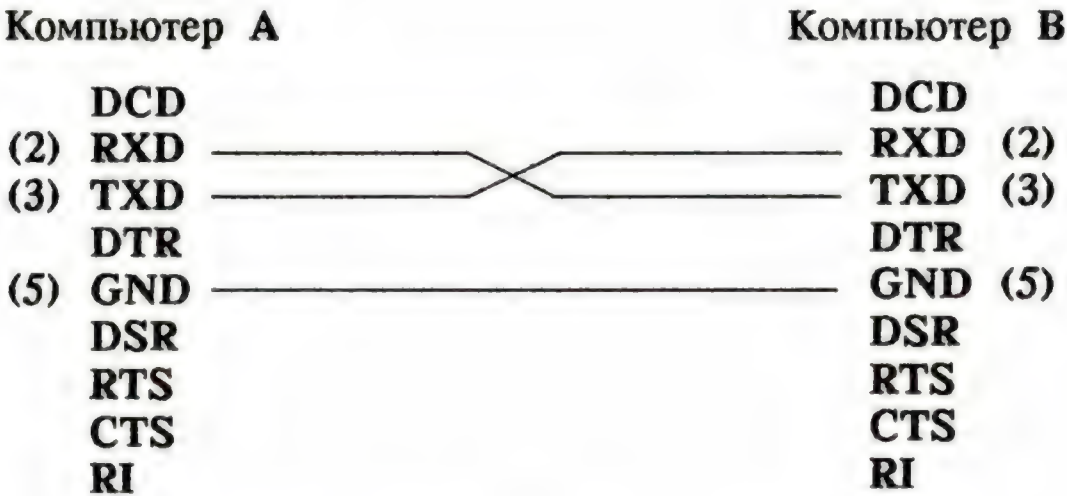


Рис. 3.1. Соединение двух компьютеров с использованием минимально возможного числа линий (в скобках приведен номер 9-контактного разъема)

В случае, когда периферийное устройство работает только на передачу или только на прием информации, иногда обходятся двумя линиями: GND и RXD или GND и TXD.

При использовании трехпроводной схемы соединения перенос информации может быть выполнен только непосредственным доступом к портам, так как в этом случае функции BIOSa не работают. Точнее, при обращении к функциям приема(передачи) символа регистрируется наличие тайм-аута. Тем не менее адаптер сохраняет способность генерировать запросы прерываний и при правильной его настройке и наличии собственного обработ-

чика прерываний может быть организована односторонняя(двухсторонняя) связь.

Трехпроводная схема для связи между компьютерами или периферийными устройствами тем не менее оставляет возможность использовать функции прерывания 14h BIOSa, но для этого придется выполнить некоторую дополнительную коммутацию контактов разъемов интерфейса друг с другом. Схема такого соединения по трем проводам приведена на рис. 3.2.

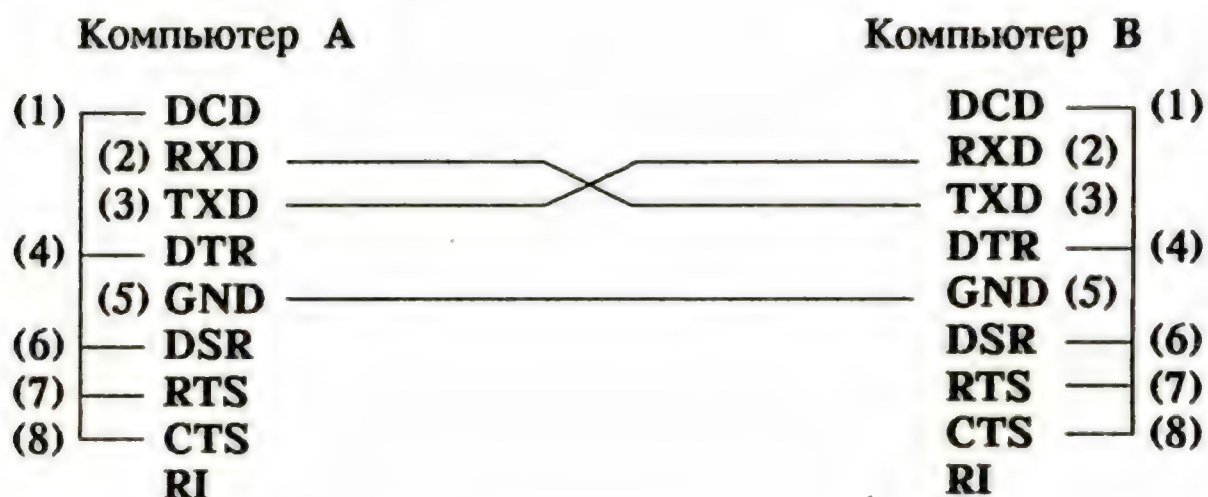


Рис. 3.2. Соединение двух компьютеров с использованием минимально возможного числа линий и сохранением всех функций BIOSa (в скобках приведен номер 9-контактного разъема)

Схемы трехпроводного соединения компьютеров требуют для надежного переноса информации организации *программного подтверждения (software handshaking)*, что снижает скорость переноса информации между компьютерами. Не-модемное соединение компьютеров друг с другом может быть выполнено и с использованием большего числа проводов по схеме, приведенной на рис. 3.3.

В рассматриваемой схеме любой из компьютеров может генерировать условие BREAK, которое будет надежно выделено другим компьютером. При необходимости инициировать передачу компьютер-передатчик может выставить на линии DCD сигнал, который будет обнаружен компьютером-приемником. Если приемник использует управляемую прерываниями асинхронную связь, то появление сигнала на линии DCD сгенерирует прерывание из-за

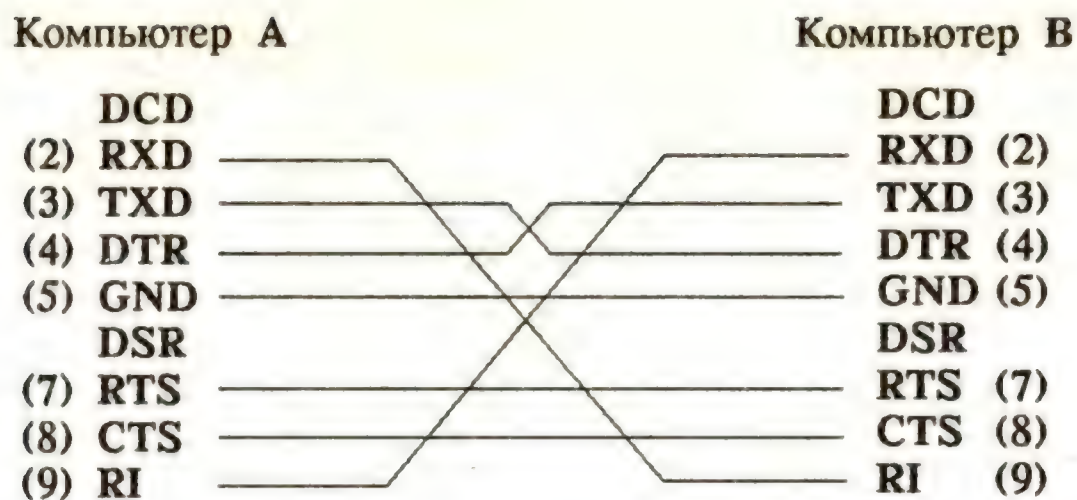


Рис. 3.3. Соединение двух компьютеров с использованием 7 линий, сохранением всех функций BIOSa и возможности аппаратного подтверждения (hardware handshaking) (в скобках приведен номер 9-контактного разъема)

изменения состояния модема. При готовности принимать данные компьютер-приемник выставляет сигнал на линии CTS. Появление этого сигнала может инициировать прерывание на передающей стороне. Подводя итог, можно сказать, что 7-проводное соединение наиболее близко к стандартному “модемному” подключению. Как результат, для переноса информации между компьютерами может использоваться программное обеспечение обслуживания модемных коммуникаций.

Для того чтобы при доступе к адаптеру асинхронной последовательной связи можно было использовать функции прерывания 14h BIOS, приходится “обманывать” адаптер, ожидающий аппаратного подтверждения сигналов. Нужно заставить адаптер “поверить” в то, что приемник всегда готов принимать данные, а передатчик всегда хочет передавать их. Достигается это соединением в разъеме интерфейса линий, как это показано на рис. 3.2. В результате такого соединения инициализированный для связи адаптер всегда получает сигналы DSR, DCD и CTS. Другими словами, адаптер считает, что другой адаптер (модем) всегда готов к приему данных от компьютера и всегда хочет передавать их.

Однако организация связи по минимально возможному числу линий между компьютерами или компьютером и периферийным устройством связана с определенными труд-

ностями. Невозможность приостановки передачи(приема) символов, например сбросом сигналов на линии CTS, DSR, DCD или RTS, делает весьма вероятной возникновение ошибки переопределения данных, которую передающий информацию компьютер просто не может обнаружить. Ведь он работает “вслепую”, не имея никакой обратной связи с приемником. Способ устранения этой проблемы при организации двухсторонней связи между компьютерами — это организация *программного подтверждения приема информации*.

3.4.2. Программное подтверждение приема

Суть способа программного подтверждения состоит в следующем:

1) компьютеры работают по принципу “ведущий-ведомый”. *Ведущий компьютер* выдает команды, а *ведомый* (или, как иногда говорят, *удаленный*) компьютер их выполняет;

2) каждый компьютер еще до начала переноса информации “знает”, какую роль он играет — передатчика или приемника информации. Если удаленный компьютер хочет передать файл или блок информации фиксированной длины, он обращается к ведущему компьютеру с запросом передачи информации. После этого ведущий компьютер настраивается на прием информации от удаленного компьютера. Если удаленный компьютер хочет прочесть файл в ведущем компьютере, он обращается к последнему с запросом на прием информации и сообщает дополнительную информацию, например имя файла, который должен передаваться. Порядок обмена такой командной информацией называют *командным протоколом*;

3) прежде всего регистрируется наличие связи между двумя компьютерами. Для этого передающий компьютер выполняет передачу в интерфейс заранее оговоренного символа, например ‘?’. Компьютер-приемник при готовности принимать информацию возвращает специальный символ, например ‘.’;

4) передача информации выполняется с использованием заранее оговоренного формата “пакета” информации. В самом простом случае информация передается просто блоками фиксированной длины, например по 512 байт. Если информация переносится целыми файлами и допускается передача как текстовых, так и двоичных файлов, довольно трудно обнаружить момент завершения передачи. Одна из возможностей — завершение передачи файла генерацией условия BREAK в передатчике. Вообще, BREAK можно использовать и для сообщения о других особых ситуациях в интерфейсе, например возникновении ошибки на приемной стороне. Частный случай команды — передача после условия BREAK символа конца файла. Приемная сторона, выделив условие BREAK, после которого следует символ 1Ah (Ctrl-Z), считает передачу файла законченной. Недостатком использования условия BREAK в трехпроводной связи является невозможность различить условие BREAK и ситуацию временного пропадания физической связи между передатчиком и приемником. Поэтому более надежным будет способ передачи размера файла в самом начале пакета. Протокол связи между файловым сервером и удаленным компьютером может предусматривать передачу также и имени файла. В самых сложных случаях формат пакета включает размер пакета в байтах, имя файла в передатчике, имя файла в приемнике, байты передаваемого файла и контрольную сумму всех байтов пакета;

5) каждый из компьютеров работает как на передачу, так и на прием. Передав символ, компьютер считывает и анализирует принятый символ. Приняв символ, компьютер возвращает его обратно в интерфейс. Компьютер-передатчик сопоставляет переданный и возвращенный символы. При их совпадении выполняется передача следующего символа, а при несовпадении фиксируется ошибка передачи. Поведение программы после обнаружения ошибки может включать:

а) игнорирование ошибки и извещение приемника об имевшихся ошибках после завершения передачи пакета;

б) простое завершение передачи; при этом приемник фиксирует ошибку тайм-аута и запрашивает передатчик о причине прекращения передачи; ответом передатчика может служить сообщение о повторении передачи последнего символа, переданного ранее с ошибкой.

3.4.3. Перенос файлов между компьютерами

Далее приводится пример утилиты для переноса файла по трехпроводному интерфейсу. Имя переносимого файла, номер используемого адаптера последовательной связи и “исполняемая” утилитой роль определяются в командной строке при запуске программы. Первый параметр задает выполняемую программой функцию. Если он — /s или /S, программа будет передавать файл, имя которого является вторым параметром командной строки через адаптер, номер которого задан третьим параметром командной строки. Если первый параметр /r или /R, компьютер принимает информацию из адаптера последовательной связи, номер которого задан третьим параметром и помещает принятую информацию в файл, имя которого задано вторым параметром командной строки. Например, запуск программы следующей командной строкой

```
COM_LINK /S D:\TC\SOURCE\COM_ISR.C 0
```

вызовет передачу файла D:\TC\SOURCE\COM_ISR.C в адаптер последовательной связи COM1. Запуск программы следующей командной строкой

```
COM_LINK /R H:\RECEIVED.C 1
```

вызовет прием файла H:\RECEIVED.C, принимаемого через адаптер последовательной связи COM2. Запуск программы COM_LINK с ключом /R выполняется на передающей стороне, а запуск программы COM_LINK с ключом /S — на приемной.

В программе используется доступ к портам как более быстрый и не зависящий от особенностей BIOSa конкретного компьютера. Однако это несколько усложняет про-

граммный код, прежде всего из-за необходимости организации контроля по тайм-ауту. Использование портов позволяет ограничиться схемой связи, приведенной на рис. 3.1 и не выполнять дополнительную коммутацию контактов в разъемах.

Передающая сторона контролирует правильность принятой информации по совпадению принятого и возвращенного приемником символов. В случае несовпадения символов программа аварийно завершается и регистрируется ошибка передачи. Как при передаче, так и при приеме время ответа (программного подтверждения) ограничивается временем выполнения TIME_OUT повторений цикла ожидания. Обнаружив условие тайм-аута, приемник и передатчик аварийно завершают свою работу.

Утилита включает головную функцию main() и ряд рассмотренных ранее функций. Две новые функции send_file() и receive_file() выполняют основную работу по переносу файла. Перечислим основные задачи, выполняемые функцией main():

- 1) проверка параметров командной строки;
- 2) проверка существования специфицированных файлов;
- 3) установление связи между компьютерами;
- 4) вызов одной из функций переноса файлов;
- 5) анализ кода возврата функции переноса и печать сообщения при аварийном завершении.

```
/* L3_8.C - COM_LINK
```

```
Программа переноса файлов по интерфейсу RS-232.
```

```
Параметры командной строки:
```

```
1-й параметр задает операцию:
```

```
    /S или /s - выполняется прием файла;
```

```
    /R или /r - выполняется передача файла;
```

```
2-й параметр задает имя файла;
```

```
3-й параметр задает адаптер:
```

```
    1 - используется COM1;
```

```
    2 - используется COM2 и т.д. */
```

```
#include <string.h>
```

```
#include <stdlib.h>
```



```
#include <fcntl.h>
#include <dos.h>
#include <bios.h>
#include <io.h>
```

```
#define SENDER 0
#define RECEIVER 1
#define TIME_OUT 0x0000ffffl
#define NOT_OPENED_FILE -100
#define NOT_ENOUGH_MEMORY -99
#define WRITE_ERROR -98
#define TIME_OUT_ERROR -97
#define COM_ERROR -96
#define KEY_PRESSED -95
#define OK 0
```

```
char * message[] = {
    "Неправильно заданы параметры командной строки\n"\
    "\x7Использование программы:\n\n "\
    "COM_LINK {/s || /S, /r || /R} "\
    "ИМЯ_ФАЙЛА НОМЕР_ПОРТА\n\nгде: "\
    "S - компьютер передает файл ИМЯ_ФАЙЛА;\n"\
    "    R - компьютер принимает информацию и "\
    "помещает ее в ИМЯ_ФАЙЛА;\n"\
    "    НОМЕР_ПОРТА = 1 для COM1; = 2 для COM2; "\
    "= 3 для COM3; = 4 для COM4.\n",
    "\x7В системе отсутствует запрошенный адаптер\n",
    "Файл существует. Переопределяете ? (Y/N)\n",
    "\x7Специфицированный файл не существует. Завершение\n",
    "Запустите программу COM_LINK "\
    "на другом компьютере с ключом /",
    "После этого нажмите любую клавишу...\n",
    "Ошибка инициализации адаптера. Завершение\n",
    "Начало цикла переноса. Завершение "\
    "процесса - нажатие любой клавиши.\n\n",
    "Установлена связь. Начинается перенос файла.\n",
    "\x7Не хватает памяти для исполнения "\
    "программы. Завершение\n",
    "\x7Ошибка записи информации на диск. Завершение\n",
    "\x7Ошибка открытия файла. Завершение\n",
    "\x7Исчезновение связи. Проверьте линию. Завершение\n",
    "\x7Ошибка приема информации. Завершение\n",
```



```

"\x7Преждевременное завершение переноса файла\n",
0,
};

/* Прототипы используемых функций. */
int com_inst(void);
unsigned send_symbol(char);
unsigned receive_symbol(void);
int send_file(char * );
int receive_file(char *);
void set_baud(long unsigned, char);

unsigned base_addr;
long unsigned time_out = TIME_OUT;

int main(int argc, char ** argv)
{
int com_port, fp, ch, mode, ret, index;
unsigned far * bios_addr = (unsigned far *) 0x00400000L;
/* Проверка правильности параметров командной строки. */
if(!(argc >= 3 && argv[1][0] == '/'
&& ( (argv[1][1] == 's' || argv[1][1] == 'S')
|| (argv[1][1] == 'r' || argv[1][1] == 'R') )
&& (argv[3][0] >= '1' && argv[3][0] <= '4'))))
{
write(2, message[0], strlen(message[0]));
exit(-1);
}
/* Существует ли запрошенный адаптер ? */
com_port = atoi(argv[3]);
if(com_port > com_inst())
{
write(2, message[1], strlen(message[1]));
exit(-2);
}
/* Проверка существования заданного файла. */
if(argv[1][1] == 'r' || argv[1][1] == 'R')
mode = RECEIVER;
else
mode = SENDER;
fp = _open(argv[2], O_RDONLY || O_BINARY);

```



```

if(mode == RECEIVER)
    if(fp != -1)
    {
        write(2, message[2], strlen(message[2]));
        while(1)
        {
            ch = bioskey(0) & 0x00ff;
            if(ch == 'N' || ch == 'n')
            {
                close(fp);
                exit(-3);
            }
            else
                if(ch == 'Y' || ch == 'y')
                    break;
        }
    }
if((mode == SENDER) && (fp == -1))
{
    write(2, message[3], strlen(message[3]));
    exit(-4);
}

```

```

/*****
 * Установление связи между компьютерами. *
 *****/
/* Приглашение на запуск программы с противоположным
ключом на другом компьютере. */
write(1, message[4], strlen(message[4]));
if(mode == SENDER)
    write(1, "R.\n", 3);
else
    write(1, "S.\n", 3);
write(1, message[5], strlen(message[5]));
bioskey(0);
/* Чтение базового адреса выбранного адаптера. */
bios_addr=(unsigned far *) (bios_addr + com_port - 1);
base_addr=*bios_addr;
/* Инициализация адаптера: 115 200 бит/с;
8 бит данных; 1 стоп-бит; четный паритет. */
set_baud(115200L, 0x03);

```


/* Попытка установления связи.

Защита по тайм-ауту не выполняется.

Время ожидания ответа не ограничено. */

write(1, message[7], strlen(message[7]));

while(1) /* выход из цикла по нажатию клавиши

или при получении '!'

*/

{

if(bioskey(1))

{

if(mode == RECEIVER)

{

close(fp);

unlink(argv[2]); /*удаление созданного файла*/

}

write(2, message[14], strlen(message[14]));

bioskey(0); /* удаление символа из

буфера клавиатуры */

return KEY_PRESSED;

}

/* Обмен символами '?' и '!'. */

if(mode == SENDER)

{

send_symbol('?');

if((receive_symbol() & 0x00ff) == '!')

{

write(1, message[8], strlen(message[8]));

break;

}

}

else

{

if(((ch = receive_symbol() & 0x00ff) == '?') &&

((ch & 0x8e00) == 0))

{

send_symbol('!');

write(1, message[8], strlen(message[8]));

break;

}

}

}


```

/*****
 * Перенос файла по интерфейсу.
 *****/
close(fp); /* функции передачи(приема)
           сами открывают файл */
if(mode == SENDER)
    ret = send_file(argv[2]);
else
    ret = receive_file(argv[2]);
/*****
 * Вывод сообщения об ошибке.
 *****/
switch(ret) {
case NOT_ENOUGH_MEMORY:
    index = 9;
    break;
case WRITE_ERROR:
    index = 10;
    break;
case NOT_OPENED_FILE:
    index = 11;
    break;
case TIME_OUT_ERROR:
    index = 12;
    break;
case COM_ERROR:
    index = 13;
    break;
case KEY_PRESSED:
    index = 14;
    bioskey(0); /* удаление символа из
                буфера клавиатуры */
    break;
default: index = 15;
        break;
}

if(mode == RECEIVER && ret != OK) unlink(argv[2]); /*
           удаление не полностью принятого файла */
write(2, message[index], strlen(message[index]));
return ret;
}

```


Передачу файла выполняет функция `send_file()`.

`/* L3_9.C`

Передает файл `file_name` в адаптер последовательной связи.

Завершение программы:

по нажатии любой клавиши на передающей стороне;

при аварийном завершении на приемной стороне;

при превышении времени ожидания ответа;

В о з в р а т :

<code>NOT_OPENED_FILE</code>	<code>-50</code>	– неудача открытия файла;
<code>NOT_ENOUGH_MEMORY</code>	<code>-99</code>	– недостаточно памяти;
<code>TIME_OUT_ERROR</code>	<code>-97</code>	– превышение времени ожидания ответа три раза подряд;
<code>COM_ERROR</code>	<code>-96</code>	– несовпадение переданного символа и принятого “эха” или ошибка адаптера;
<code>KEY_PRESSED</code>	<code>-95</code>	– аварийное завершение по нажатию любой символьной клавиши;
<code>OK</code>	<code>0</code>	– нормальное завершение программы. <code>*/</code>

```
#include <stdio.h>
#include <stat.h>
#include <fcntl.h>
#include <io.h>
#include <stdlib.h>
#include <bios.h>
#include <dos.h>
```

```
#define NOT_OPENED_FILE    -50
#define NOT_ENOUGH_MEMORY -49
#define TIME_OUT_ERROR    -97
#define COM_ERROR         -96
#define KEY_PRESSED       -95
#define OK                 0
```

`/* Прототипы используемых функций. */`

`unsigned receive_symbol(void);`

`unsigned send_symbol(char);`

`void prn_repeat(char, int);`


```

int send_file(char * file_name)
{
    unsigned far * head_ptr = (unsigned far *) 0x0000041AL;
    unsigned far * tail_ptr = (unsigned far *) 0x0000041CL;
    int fp, i, first_time = 1, number, repeat = 0,
        ret_send, ret_receiv;
    struct BUF {
        long size, buffer[4096];
    } * size_buffer;
    char * start;
    struct stat work;
    /* Открытие передаваемого файла. */
    if((fp = _open(file_name, O_RDONLY || O_BINARY)) == -1)
        return NOT_OPENED_FILE;
    /* Выделение необходимого объема памяти. */
    if((size_buffer = (struct BUF *) malloc(4100)) == NULL)
        return NOT_ENOUGH_MEMORY;
    /* Определение длины передаваемого файла. */
    if(fstat(fp, &work) == -1)
        return NOT_OPENED_FILE;
    else
        size_buffer->size = work.st_size + 4;
    /* Вывод строки визуализации процесса переноса файла.
       Один прямоугольник соответствует 4096 байтам. */
    printf("Передача файла %s\n", file_name);
    prn_repeat(0xb1, (work.st_size >> 12) + 1); /* вывод
                                                    строки */
    /* Передача размера файла первыми четырьмя символами,
       а затем остальных символов файла. Размер файла
       увеличивается на 4. */
    while((number = read(fp, size_buffer->buffer,
        4096)) != 0)
    {
        /* Цикл посимвольной передачи. */
        if(first_time)
        {
            first_time = 0;
            start = (char *) size_buffer;
            number += 4;
        }
        else

```



```

        start = (char *) (size_buffer -> buffer);
    for(i = 1; i <= number; i ++)
    {
        ret_send = send_symbol(*start);
        while(1) /* выход из цикла либо по
                    нажатию клавиши, либо по
                    получению не тайм-аута */
        {
            if(*head_ptr != *tail_ptr)
            {
                close(fp);
                return KEY_PRESSED;
            }
            /* завершение по нажатию клавиши */
            ret_receiv = receive_symbol();
            if(!(ret_receiv & 0x8000))
                /* не тайм-аут */
                break;
            else
                repeat ++;
            if(repeat == 3)
            {
                close(fp);
                return TIME_OUT_ERROR;
                /* возврат по тайм-ауту */
            }
        }
        if((ret_receiv & 0x00ff) ==
            (ret_send & 0x00ff))
        {
            start ++;
            repeat = 0;
        }
        else
        {
            close(fp);
            return COM_ERROR; /* ошибка
                                на приемной стороне */
        }
    }
    putchar(0x00db); /* отображение
                        процесса переноса */

```



```

    }
    close(fp);
    free(size_buffer);
    return OK;
}

```

Прием символа выполняет функция `receive_file()`.

`/* L3_10.C`

Принимает файл из адаптера последовательной связи и помещает его под именем `file_name`.

Завершение программы:

по нажатии любой клавиши на приемной стороне;
 при аварийном завершении на передающей стороне;
 при превышении времени ожидания ответа;

В о з в р а т :

<code>NOT_OPENED_FILE</code>	-100	- неудача открытия файла;
<code>NOT_ENOUGH_MEMORY</code>	-99	- недостаточно памяти;
<code>WRITE_ERROR</code>	-98	- ошибка при записи в файл;
<code>TIME_OUT_ERROR</code>	-97	- превышение времени ожидания ответа;
<code>COM_ERROR</code>	-96	- ошибка адаптера;
<code>KEY_PRESSED</code>	-95	- аварийное завершение по нажатию любой символьной клавиши;
<code>OK</code>	0	- нормальное завершение программы. */

```

#include <stdio.h>
#include <stat.h>
#include <fcntl.h>
#include <io.h>
#include <stdlib.h>
#include <bios.h>
#include <dos.h>

```

```

#define NOT_OPENED_FILE -100
#define NOT_ENOUGH_MEMORY -99
#define WRITE_ERROR -98
#define TIME_OUT_ERROR -97
#define COM_ERROR -96
#define KEY_PRESSED -95
#define OK 0

```



```

/* Прототипы используемых функций. */
unsigned receive_symbol(void);
unsigned send_symbol(char);
void prn_repeat(char, int);

int receive_file(char * file_name)
{
    unsigned far * head_ptr = (unsigned far *) 0x0000041AL;
    unsigned far * tail_ptr = (unsigned far *) 0x0000041CL;
    int fp, number, ret_receiv;
    long total = 0;
    struct BUF {
        long size, buffer[4096];
    } * size_buffer;
    char * start;
    /* Открытие принимаемого файла. */
    if((fp = open(file_name,
        O_CREAT | O_WRONLY | O_TRUNC | O_BINARY,
        S_IWRITE)) == -1)
        return NOT_OPENED_FILE;
    /* Динамическое распределение памяти
       под приемный буфер. */
    if((size_buffer = (struct BUF *) malloc(4100)) == NULL)
        return NOT_ENOUGH_MEMORY;
    printf("Прием файла %s\n", file_name);
    size_buffer -> size = 0xffff; /* первые 4 принятых
                                   байта переопределяют это значение */
    start = (char *) size_buffer;
    number = 4100;
    while(total++ < size_buffer -> size) /* выход после
        передачи всех символов или по нажатии клавиши*/
    {
        /* Цикл посимвольного приема. */
        {
            if(*head_ptr != *tail_ptr)
            {
                close(fp);
                return KEY_PRESSED; /*выход из цикла по
                                     нажатии клавиши*/
            }
            ret_receiv = receive_symbol();

```



```

        if((ret_receiv & 0x8000)) /* тайм-аут */
        {
            close(fp);
            return TIME_OUT_ERROR; /* возврат по
                                    тайм-ауту */
        }
        if(ret_receiv & 0x0C00)
        {
            close(fp);
            return COM_ERROR; /* возврат по ошибке
                                приема */
        }
    }
    send_symbol(ret_receiv & 0x00ff);
    * start++ = (char) ret_receiv;
    /* После получения размера файла выводится
       строка символов, отображающих длину файла,
       и корректируется число принимаемых символов.*/
    if(total == 4)
        prn_repeat(0xb1,
                    (size_buffer -> size >> 12) + 1);
    /* После заполнения буфера размером 4096 байт
       выполняется его отгрузка на диск
       в специфицированный файл. */
    number--;
    if(!number)
    {
        if(write(fp,
                 (char *) (size_buffer -> buffer),
                 4096) <= 0)
            return WRITE_ERROR;
        putchar(0x00db); /* отображение процесса
                           переноса файла */
        number = 4096;
        start = (char *) size_buffer -> buffer;
    }
}

if((number != 0) && (number != 4096)) /* Есть ли
                                         неполный (4096 байт) буфер ? */
{
    if(write(fp,
             (char *) (size_buffer -> buffer),

```



```

        4096 - number) == -1)
            return WRITE_ERROR;
        putchar(0x00db); /* отображение процесса переноса*/
    }
    close(fp);
    free(size_buffer);
    printf("\nИз интерфейса ожидалось %ld байт." \
        " Принято %ld байт\n",
        size_buffer -> size, --total);
    return OK;
}

```

Передачу символа выполняет функция `send_symbol()`.

```

/* L3_11.C
   Посылает символ в адаптер последовательной связи.
   В о з в р а т:
   unsigned:
       старший байт – байт состояния линии, как и для
                       аналогичной функции BIOSa;
       младший байт – переданный символ.
   В н е ш н и е   п е р е м е н н ы е:
       unsigned base_addr      – базовый адрес адаптера;
       long unsigned time_out  – число повторений цикла
                               ожидания пустого регистра
                               данных передатчика. */

```

```

#include <dos.h>

```

```

unsigned send_symbol(char symbol)
{
    extern unsigned base_addr;
    extern long unsigned time_out;
    register unsigned word = 0, tick = 0,
                        status, base_addr_plus_5;
    base_addr_plus_5 = base_addr + 5;
    word = symbol;
    while(tick++ < time_out)
    {
        status = inportb(base_addr_plus_5);
        if(status & 0x20)
        {

```



```

        outportb(base_addr, symbol);
        return word | (status << 8);
    }
}
return word | ((status << 8) | 0x8000);
}

```

Прием символа из адаптера выполняет функция `receive_symbol`.

```

/* L3_12.C
  Читает символ из адаптера последовательной связи.
  В о з в р а т:
  unsigned:
    старший байт – байт состояния линии, как и
                      для аналогичной функции BIOS;
    младший байт – прочитанный символ.
  В н е ш н и е   п е р е м е н н ы е:
    unsigned base_addr – базовый адрес адаптера.
    long unsigned time_out – число повторений цикла
                          ожидания готовности данных приемника. */
#include <dos.h>
unsigned receive_symbol(void)
{
    extern unsigned base_addr;
    extern long unsigned time_out;
    register unsigned word = 0, tick = 0,
                      status, base_addr_plus_5;
    base_addr_plus_5 = base_addr + 5;
    while(tick++ < time_out)
    {
        status = inportb(base_addr_plus_5);
        if(status & 0x01)
        {
            word = inportb(base_addr);
            return word | (status << 8);
        }
    }
    return word | ((status << 8) | 0x8000);
}

```

Вспомогательная функция `prn_repeat()` используется для визуализации хода процесса переноса файла.


```
/* L3_13.C
```

```
    Вывод строки одинаковых символов symbol с атрибутом  
    attr n раз с текущей позиции курсора без его сдвига.*/  
#include <dos.h>  
void prn_repeat(char symbol, char attr, int n)  
{  
    union REGS r;  
    r.x.cx = n;  
    r.h.ah = 0x09;  
    r.h.al = symbol;  
    r.x.bx = attr;  
    int86(0x10, &r, &r); /* вывод строки символов  
                           без сдвига курсора */  
}
```

Приведенная программа обеспечивает скорость переноса файлов между компьютерами, сравнимую со скоростью секции связи (Link) оболочки Norton Commander, но в отличие от нее обеспечивает более защищенный перенос файлов.

Возможные улучшения рассмотренной программы сводятся к следующему:

1) переработав функцию main(), можно организовать групповой перенос файлов, имена которых удовлетворяют символам-шаблонам '*' и '?'. В этом случае функция передачи файлов вызывается до тех пор, пока в специфицированной директории имеются файлы, имена которых соответствуют заданной спецификации;

2) переработав функции переноса файлов, можно дополнительно переносить и имя передаваемого файла или его полную спецификацию, включая букву накопителя и маршрут на передающей стороне;

3) как на передающей, так и на приемной стороне можно распределять буфер не фиксированного размера (4100 байт), а максимального, как это сделано в примере программы максимально быстрого копирования файлов (см. вторую книгу комплекса). Возможны и многие другие доработки рассматриваемой программы.

4. ВВОД ИНФОРМАЦИИ С МАНИПУЛЯТОРА “МЫШЬ”

4.1. Аппаратные средства манипулятора “мышь”

Манипулятор “мышь” является одним из самых распространенных периферийных устройств персонального компьютера. Профессиональное программное обеспечение в настоящее время не имеет шансов на коммерческий успех при отсутствии поддержки ввода информации с манипулятора. В данной главе рассматриваются аппаратные и программные средства манипулятора и приводятся примеры программирования для ввода информации с помощью “мыши”.

Аппаратно манипулятор представляет собой периферийное устройство, подключаемое к одному из адаптеров последовательной связи компьютера. В некоторых персональных компьютерах манипулятор может подключаться к специально спроектированному адаптеру — так называемому *порту позиционирующего устройства (pointing device port)*. Принципиально логика работы манипулятора от этого не зависит, поэтому в данной главе рассматривается манипулятор, подключаемый к адаптеру последовательной связи. Формат информации, передаваемой в адаптер, зависит от типа манипулятора, а точнее, от режима его работы. В настоящее время наибольшее распространение получили два основных режима работы манипуляторов:

1) режим Mouse mode, в котором манипулятор имеет три управляющие кнопки;

2) режим Microsoft mode, в котором манипулятор имеет только две управляющие кнопки.

Достаточно распространены “мыши”, способные переключаться из одного режима в другой. Переключение из режима в режим выполняется либо аппаратно с помощью специальной дополнительной кнопки (например, манипулятор Genius Mouse), либо реализуется механизм программного переключения (так называемый softswitch): выбор режима работы выполняет драйвер “мыши”. Если в момент загрузки драйвера в память нажата, например, левая кнопка, выбирается режим Microsoft Mouse, если же кнопка не нажата — режим Mouse mode.

В состав манипулятора входят датчики перемещения “мыши” в горизонтальном и вертикальном направлениях, индикатор нажатия кнопок и блок связи с интерфейсом RS-232 (рис. 4.1).

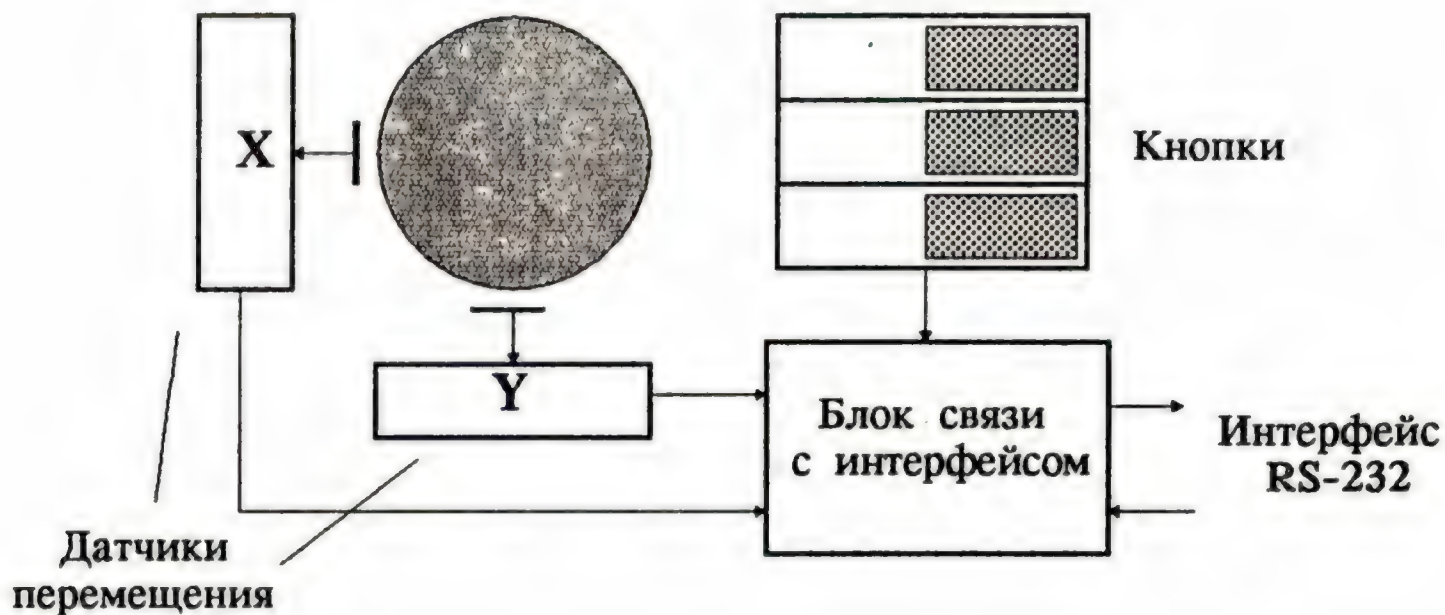


Рис. 4.1. Схема построения манипулятора “мышь”.

Блок связи с интерфейсом передает в адаптер последовательной связи байты данных при любом нажатии кнопок (buttons) и перемещении “мыши” в горизонтальном или вертикальном направлении на определенную

величину, большую так называемого “микки” (*Mickey*). Один “микки” обычно равен 1/200-й части дюйма. Есть манипуляторы и с более высокой чувствительностью датчика перемещения: 1/320-я или даже 1/400-я дюйма. Передаваемые в адаптер значения смещения имеют “знак”. Например, при перемещении манипулятора снизу вверх на один “микки” передается байт 01h, а при перемещении сверху вниз — байт FFh. Обработчик прерываний от адаптера последовательной связи суммирует получаемые значения с учетом знака. Таким образом, всегда имеется возможность определить “текущую” позицию манипулятора относительно момента начала наблюдения за “мышью”. Обычно “наблюдение” за “мышью” начинается после загрузки драйвера “мыши” или выполнения операции инициализации “мыши” из программы пользователя.

4.2. Программная поддержка манипулятора. Основные компоненты драйвера “мыши”

Аппаратно манипулятор “мышь” весьма прост, но его использование требует загрузки специальной программы — так называемого драйвера “мыши”. Он, как и любая программа-драйвер, зависит от типа обслуживаемой аппаратуры и ее особенностей:

- 1) от типа адаптера, к которому подключен манипулятор;
- 2) от режима, в котором работает манипулятор;
- 3) от формата слов и частоты передачи информации при подключении к адаптеру последовательной связи.

Манипулятор всегда поставляется со своим драйвером, разрабатываемым изготовителем “мыши”. Драйвер может поставляться в двух формах: как устанавливаемый драйвер (.SYS-файл) и как .EXE- или .COM-файл. В первом случае драйвер помещается в память в процессе начальной загруз-

ки системы на этапе обработки файла CONFIG.SYS. Во втором случае драйвер представляет собой обычную TSR-программу (подробнее о TSR см. в гл. 2). Выполняемые драйвером действия не зависят от формата его файла, а различия между двумя этими формами объясняются специфическим форматом устанавливаемого драйвера устройств в MS-DOS.

Основными частями драйвера “мыши” являются:

- 1) секция инсталляции драйвера;
- 2) обработчик аппаратных прерываний от адаптера последовательной связи. В компьютере IBM PC XT адаптерам COM1 и COM2 соответствуют прерывания 0Bh и 0Ch, в IBM PC AT — 0Ch и 0Bh. (О закреплении аппаратных прерываний за внешними устройствами подробнее см. в 1.3 второй книги комплекса);
- 3) обработчик программного прерывания интерфейса прикладной программы с манипулятором “мышь” (прерывание 33h);
- 4) обработчик программного прерывания 10h управления экраном.

Секция инсталляции проверяет, подключена ли физически “мышь” нужного типа и не был ли драйвер установлен ранее. Если он не был установлен, инсталлятор “подменяет” вектор прерывания 0Ch или 0Bh точкой входа в собственный обработчик прерывания. “Старый” обработчик прерывания, если он был, теряется для системы. По умолчанию всегда используется COM1, но многие драйверы позволяют явно задавать в командной строке номер адаптера, к которому подключен манипулятор. Затем инсталлятор настраивает аппаратуру адаптера на генерацию запросов прерываний, деблокирует в контроллере прерываний биты, соответствующие нужной линии (IRQ3 или IRQ4), устанавливает нужный формат слова и частоту передачи. Кроме этого, инсталлятор подменяет вектор прерывания 10h адресом собственного обработчика. “Старый” обработчик прерывания 10h включается по каскадной схеме. Основным объемом программы драйвера со-

ставляет код обработчика прерывания 33h, точку входа в который инсталлятор помещает в таблицу векторов прерываний. Ряд внутренних переменных, доступных через прерывание 33h, устанавливается инсталлятором в значение по умолчанию. Последним действием инсталлятора является резидентное завершение программы драйвера.

Обработчик прерывания от адаптера принимает байты данных от манипулятора и изменяет те или иные внутренние переменные: текущие относительные позиции по горизонтали и вертикали, текущее состояние кнопок манипулятора (нажата или отпущена) и т.д. Кроме этих действий, обработчик прерывания выполняет перемещение по экрану курсора манипулятора. Форма курсора зависит от типа видеорежима (текстовый или графический) и может изменяться через прерывание 33h.

Необходимость перемещения курсора по экрану и зависимость этих действий от режима работы видеоадаптера вынуждают “перехватывать” прерывание 10h. Драйвер “мыши” отслеживает все переключения режимов и изменения во внутренних регистрах видеоадаптера, выполняемые функциями прерывания 10h BIOS. Например, драйвер изменяет свою внутреннюю переменную, отражающую текущий режим адаптера, если запрашивается функция AH = 00h переключения видеорежима. Выполнив все дополнительные действия, “новый” обработчик прерывания 10h вызывает “старую” ISR 10h, которая выполняет запрошенные действия по управлению экраном.

Обработчик прерывания 33h представляет собой коллекцию функций, образующих интерфейс прикладной программы с драйвером манипулятора. Число функций зависит от типа драйвера “мыши”. Номер функции задается в регистре AH. Если для выбранной функции требуются дополнительные параметры, они помещаются в других регистрах процессора. Возвращаемые значения также передаются в точку вызова в регистрах. (Подробно функции прерывания 33h рассматриваются далее.)

4.3. Основные функции интерфейса прикладной программы с манипулятором “мышь”

4.3.1. Подготовительные операции программы

Для работы по управлению манипулятором и организации ввода информации прикладная программа выполняет различные функции прерывания 33h. В настоящее время де-факто большинство драйверов “мыши” поддерживают стандарт, введенный фирмой Microsoft для различных функций прерывания 33h. Многие системы программирования имеют в своем составе библиотечные функции, выполняющие обращение к той или иной функции прерывания 33h. Библиотечные функции работают медленнее, чем непосредственное обращение к прерыванию 33h из прикладной программы. Кроме того, многие программисты в нашей стране используют неполные или даже несанкционированные копии систем программирования, в которых библиотеки поддержки манипулятора отсутствуют или работают некорректно. Поэтому в дальнейшем приводятся примеры непосредственного обращения к прерыванию 33h, а не библиотечные функции Turbo C.

Каждая программа, использующая для ввода информации манипулятор “мышь”, должна выполнить ряд подготовительных операций:

- 1) определить, установлен ли драйвер “мыши”;
- 2) задать вид и форму курсора манипулятора;
- 3) описать границы перемещения курсора “мыши” по экрану;
- 4) описать “чувствительность” курсора, равную числу “микки”, приходящихся на один пиксел экрана по горизонтали и вертикали;
- 5) установить порог “удвоенной” скорости курсора “мыши”;
- 6) “включить” курсор манипулятора (сделать его видимым на экране);

7) установить курсор в начальную позицию на экране в соответствии с нуждами программы.

Многие из этих действий помогает осуществить функция инициализации и определения текущего состояния драйвера манипулятора (AX = 0000h). Ниже приводится спецификация этой функции.

При вызове:

AX = 0000h.

Возврат:

AX — состояние оборудования:

- 0000h — отсутствует драйвер “мыши” (отсутствует аппаратура, не загружен драйвер);

- FFFFh — “мышь” готова к работе;

BX — количество кнопок манипулятора:

- 0000h — не две кнопки;

- 0002h — две кнопки (режим Microsoft mode);

- 0003h — три кнопки (режим Mouse Systems mode).

При выполнении функции AX = 0000h драйвер манипулятора приводится в состояние по умолчанию:

- 1) курсор манипулятора установлен в центре экрана и выключен;

- 2) чувствительность манипулятора по вертикали равна 2 микки/пиксел, по горизонтали — 1 микки/пиксел;

- 3) порог удвоенной скорости установлен равным 64 микки/с;

- 4) установлена форма курсора по умолчанию;

- 5) координаты левого верхнего угла области перемещения курсора манипулятора соответствуют координатам (0, 0), а координаты нижнего правого угла области перемещения манипулятора — максимальным координатам текущего видеорежима минус 1.

Если необходимо установить значения, отличающиеся от значений по умолчанию, используются функции установки формы курсора, границ его перемещения, задания значений чувствительности и порога удвоенной скорости.

Для определения того, работоспособна ли “мышь”, необходимо выполнить функцию AX = 0000h и проанализировать возврат в AX. Если AX = -1, это указывает на готовность манипулятора к работе. Здесь следует сделать два замечания. Первое из них состоит в том, что опера-

ционная система MS-DOS “цепляет” при загрузке за вектор прерывания 33h “заглушку” из единственной инструкции IRET. Поэтому выполнение инструкции INT 33h даже при отсутствии драйвера не порождает никаких проблем. Второе замечание заключается в том, что успешное завершение инициализации не означает, однако, что аппаратура адаптера или самого манипулятора исправна, а свидетельствует только о том, что загружен драйвер “мыши”. В ряде случаев драйвер “мыши” передает в манипулятор заранее заданную последовательность слов информации. Эти слова вызывают в манипуляторе генерацию ответного “эха”, анализ которого позволяет драйверу однозначно опознать “свою” аппаратуру манипулятора и проверить исправность цепей приема информации адаптера асинхронной последовательной связи.

Приводимый далее фрагмент Си-кода иллюстрирует использование функции инициализации манипулятора “мышь”.

```
...
/* Инициализация “мыши”. */
_AX=0;
geninterrupt(0x33);
if (_AX==0)
{
    puts("НЕТ “мыши” !");
    exit(-1);
}
...
```

Успешное завершение инициализации позволяет продолжить работу прикладной программы по подготовке к использованию “мыши”. Следующая операция связана с определением типа и формы курсора. Драйверы “мыши” в текстовом режиме работы видеоадаптера поддерживают два типа курсора:

1) “жесткий”, совпадающий с обычным курсором текстового режима в форме нескольких сплошных телевизионных строк в пределах знакоместа;

2) программируемый, представляющий собой знакоместо с измененным атрибутом символа и, возможно, со спе-

цифицированным пользователем символом. Курсор “мыши” в текстовом режиме перемещается по знакоместам экрана. Выбор типа и параметров курсора “мыши” в текстовом режиме работы видеоадаптера выполняется с помощью функции $AX = 0Ah$ прерывания $33h$, спецификация которой приводится далее.

При вызове:

$AX = 000Ah$ — установка курсора “мыши” в текстовом режиме;

BX — выбор типа курсора:

— 00 — программируемый курсор;

— 01 — “жесткий” курсор;

CX — AND-маска (маска экрана) для программируемого курсора или номер верхней скэн-линии для “жесткого” курсора;

DX — XOR-маска (маска курсора) для программируемого курсора или номер нижней скэн-линии для “жесткого” курсора.

Возврат: нет.

Если выбран “жесткий” курсор, то курсор “мыши” на экране имеет форму обычного текстового. Преимуществом такого курсора является то, что на экране присутствует только один привычный курсор и при любой операции ввода информации с использованием функций MS-DOS (например, функций стандартного ввода-вывода) курсор “мыши” автоматически перемещается в текущую текстовую позицию BIOSa. Это не означает, однако, что драйвер “мыши” при перемещении курсора обновляет слово текущей позиции курсора (см. 8.6 второй книги комплекса) в области данных BIOSa. Поэтому без дополнительных усилий со стороны программиста не удастся поместить символ на экране в позицию, на которую указывает курсор “мыши”.

При формировании программируемого текстового курсора используется: 1) слово видеопамати, образованное символом и атрибутом знакоместа экрана, в котором находится курсор манипулятора — `screen_word`; 2) AND-маска — `AND_mask`; 3) XOR-маска — `XOR_mask`. Результирующее представление курсора на экране формируется поразрядными логическими операциями по формуле

$$\text{screen_word} = \text{screen_word} \text{ AND } \text{AND_mask} \text{ XOR } \text{XOR_mask}$$

При перемещении курсора в другую позицию предыдущее содержимое видеобуфера восстанавливается. Таким образом, пользователь может установить различные формы программируемого текстового курсора “мыши”. Например, задание `AND_mask = 00FFh` и `XOR_mask = xx00h` приведет к тому, что текстовый курсор будет прямоугольником, сохраняющим любой предыдущий символ, атрибут которого `xx`. Младшая шестнадцатеричная цифра будет задавать цвет контура символа, а старшая — цвет фона. При этом, однако, возможно “исчезновение” курсора, если атрибут символа уже был равен `xx`. Распространенным является программируемый текстовый курсор со следующими значениями: `AND_mask = FFFFh` и `XOR_mask = 7700h`. В этом случае инвертируется как цвет фона, так и цвет атрибута, что гарантирует видимость курсора и не изменяет символ в позиции курсора. Нетрудно добиться того, чтобы курсор “мыши” на экране отображался в виде нужного символа. Например, для того чтобы текстовый курсор “мыши” имел форму “птички” (ASCII-код `FBh`), необходимо задать такие маски: `AND_mask = 0000h` и `XOR_mask = xxFBh`, где `xx` задает атрибут символа курсора. Общеупотребительной является практика изменения формы текстового курсора для индикации обнаруженного нажатия той или иной кнопки манипулятора “мышь”.

Далее приводится текст Си-функции задания параметров курсора манипулятора “мышь” для текстового режима работы адаптера.

```
/*
```

```
L4_1.C
```

```
    Установка заданной формы курсора в текстовом режиме.
    Параметр int mode — выбирает либо “жесткий” (0), либо
    программируемый курсор (1). Если выбран программируе-
    мый курсор, AND_mask задает маску экрана, а XOR_mask —
    маску курсора. В противном случае AND_mask задает
    номер верхней скэн-линии, а XOR_mask — номер нижней
    скэн-линии. */
```

```
#include <dos.h>
```



```

void set_text_cursor(int mode, unsigned int AND_mask,
                    unsigned int XOR_mask)
{
    struct REGPACK r;
    r.r_ax = 0x0a;
    r.r_bx = mode;
    r.r_cx = AND_mask;
    r.r_dx = XOR_mask;
    intr(0x33, &r);
}

```

В графическом режиме работы видеоадаптера может быть описана любая собственная форма курсора в пределах прямоугольника 16 x 16 пикселей. Курсор в графическом режиме описывается двумя масками размером 16 x 16 бит каждая. Одна из масок называется *AND-маской* (маской экрана), другая — *XOR-маской* (маской курсора). При перемещении курсора предыдущее содержимое экрана восстанавливается драйвером “мыши”. Маски, комбинируясь, определяют способ обработки текущего кода цвета пиксела на экране. Результаты обработки представлены в табл. 4.1.

Табл. 4.1. Формирование кода цвета пикселей изображения курсора “мыши” в графическом режиме

Бит AND-маски	Бит XOR-маски	Цвет пиксела на экране
0	0	Цвет фона
0	1	Белый цвет
1	0	Текущий цвет пиксела
1	1	Побитовая инверсия текущего цвета пиксела

С помощью различных масок можно установить любую форму курсора и добиться его видимости на любом фоне. Для режимов 4, 5 и 13h используются только четные столбцы в битовых масках; в противном случае курсор при использовании матрицы 16 x 16 имел бы слишком большие размеры. Для остальных режимов “прореживание” столбцов масок курсора не выполняется.

В графических режимах, кроме формы курсора, описывается так называемое “горячее пятно” (*hot spot*) в относительных координатах, за точку отсчета которых принят верхний левый угол прямоугольника 16 x 16 пикселей. “Горячее пятно” — это тот пиксел, на который указывает в данный момент графический курсор. Описание курсора манипулятора “мышь” в графическом режиме выполняет функция $AX = 9$ прерывания 33h. Далее приводится ее спецификация.

При вызове:

$AX = 0009h$ — установка курсора “мыши” в графическом режиме;

BX — номер столбца “горячего пятна” курсора относительно верхнего левого угла прямоугольника курсора;

CX — номер строки “горячего пятна” курсора относительно верхнего левого угла прямоугольника курсора;

$ES:DX$ — указатель на 32 слова масок: первые 16 слов образуют AND-маску, следующие 16 слов — XOR-маску.

Использование битов этих масок иллюстрирует табл. 4.1.

Возврат: нет.

Приведем текст Си-функции установки графического курсора манипулятора “мышь”:

```
/* L4_2.C
```

```
    Установка заданной формы курсора в графическом режиме.
    Параметры char hot_spot_row, char hot_spot_col —
    относительные координаты “горячего пятна” курсора
    “мыши” в прямоугольнике курсора; unsigned int far *
    screen_and_cursor_masks — указатель на две подряд
    расположенные маски экрана и курсора; первые 16 слов
    массива образуют маску экрана, следующие 16 — маску
    курсора. */
```

```
#include <dos.h>
```

```
void set_graph_cursor(int hot_spot_row,
                      int hot_spot_column,
                      unsigned int far * screen_and_cursor_masks)
{
    struct REGPACK r;
    r.r_ax = 9;
    r.r_bx = hot_spot_column;
    r.r_cx = hot_spot_row;
```



```

r.r_es = FP_SEG(screen_and_cursor_masks);
r.r_dx = FP_OFF(screen_and_cursor_masks);
intr(0x33, &r);
}

```

Для получения графического курсора, видимого на любом фоне, поступают следующим образом. Маску экрана (AND-маску) задают состоящей из слов FFFFh, а маску курсора (XOR-маску) описывают так, чтобы в ней стояли единицы в тех битах, которые образуют очертание курсора, а нули — во всех остальных битах маски. В качестве примера приводится вызов функции `set_graph_cursor()` для задания графического курсора в форме наклонной стрелки. Формирование битовой карты этого курсора поясняет рис. 4.2.

	2 c/c	16 c/c
	1000 0000 0000 0000	0x8000
	1100 0000 0000 0000	0xC000
	1010 0000 0000 0000	0xA000
	1001 0000 0000 0000	0x9000
	1010 1000 0000 0000	0xA800
	1011 0100 0000 0000	0xB400
	1011 1010 0000 0000	0xBA00
	1011 1101 0000 0000	0xBD00
	1011 1110 1000 0000	0xBE80
	1011 1111 0100 0000	0xBF40
	1011 1111 1010 0000	0xBF80
	1011 1111 1101 0000	0xBFCD
	1011 1111 1110 0000	0xBFED
	1010 1110 0000 0000	0xAE00
	1010 0011 0000 0000	0xA300
	1110 0011 0000 0000	0xE300

Рис. 4.2. XOR-маска отображения курсора в виде стрелки

```

...
static unsigned screen_and_cursor_masks[] = {
/* AND-маска */
0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff,
0xffff, 0xffff, 0xffff, 0xffff,

```



```

/* XOR-маска */
0x8000, 0xC000, 0xA000, 0x9000,
0xA800, 0xB400, 0xBA00, 0xBD00,
0xBE80, 0xBF40, 0xBFA0, 0xBF00,
0xBFE0, 0xAE00, 0xA300, 0xE300,
};

...
/* Инициализация курсора "мыши" в графическом режиме.
   "Горячее пятно" – левый верхний угол
   прямоугольника курсора. */
set_graph_cursor(0, 0, screen_and_cursor_masks);
...

```

Для получения белого курсора на текущем фоне можно задать следующие маски:

```

static unsigned screen_and_cursor_masks[] = {
/* AND-маска */
0x7fff, 0x3fff, 0x5fff, 0x6fff,
0x57ff, 0x4bff, 0x45ff, 0x42ff,
0x417f, 0x40bf, 0x405f, 0x402f,
0x401f, 0x51ff, 0x5cff, 0x1cff,
/* XOR-маска */
0x8000, 0xC000, 0xA000, 0x9000,
0xA800, 0xB400, 0xBA00, 0xBD00,
0xBE80, 0xBF40, 0xBFA0, 0xBF00,
0xBFE0, 0xAE00, 0xA300, 0xE300,
};

```

В последнем случае AND-маска представляет собой инвертированные слова XOR-маски: там, где в XOR-маске стоят единицы, в AND-маске стоят нули, и наоборот. Естественно, что на белом фоне возможна ситуация "невидимого" курсора.

Сразу после инициализации экрана областью допустимых перемещений курсора по умолчанию принимается весь экран. Однако всегда имеется возможность ограничить перемещение курсора отдельно по горизонтали и по вертикали. Для этого программа использует функции $AX = 0007$ и $AX = 0008$ прерывания 33h.

При вызове:

AX = 0007h — установка вертикальных границ перемещения
(ограничивают горизонтальное перемещение
курсора “мыши” по экрану);

AX = 0008h — установка горизонтальных границ перемещения
(ограничивают вертикальное перемещение
курсора “мыши” по экрану);

CX — минимальная граница перемещения курсора “мыши”;

DX — максимальная граница перемещения курсора “мыши”.

Возврат: нет.

При работе в графическом режиме границы перемещения задаются в пикселах. Для текстового режима они задаются в “виртуальных” пикселах (исходят из того, что знакоместо независимо от режима и типа адаптера занимает матрицу 8 x 8 условных пикселей). Поэтому для текстового режима 25 x 80 максимально возможными границами перемещения по вертикали (функция AX = 0008h) являются значения CX = 0 и DX = 199, а по горизонтали (функция AX = 0007h) — CX = 0 и DX = 639. Для текстового режима 50 x 80 (VGA- и MCGA-адаптеры) предельные значения таковы: по вертикали (функция AX = 0008h) CX = 0 и DX = 399, по горизонтали (функция AX = 0007h) CX = 0 и DX = 639. Если минимальная граница больше максимальной, функции меняют местами значения в регистрах CX и DX. Если в момент установки новой границы курсор манипулятора находится вне задаваемого диапазона перемещения, он “прыгает” к ближайшей из границ.

Приведем Си-функции для установки границ перемещения курсора по вертикали и горизонтали.

```
/* L4_3.C
```

```
Устанавливает начальную и конечную вертикальные границы  
(start_hor_movement, end_hor_movement) перемещения  
курсора по горизонтали. */
```

```
#include <dos.h>
```

```
void hor_movement(int start_hor_movement,  
                  int end_hor_movement)
```

```
{ _AX = 7;  
  _CX = start_hor_movement;  
  _DX = end_hor_movement;  
  geninterrupt(0x33); }
```



```

/* L4_4.C
Устанавливает начальную и конечную горизонтальные
границы (start_vert_movement, end_vert_movement)
перемещения курсора по вертикали. */

```

```

#include <dos.h>

```

```

void vert_movement(int start_vert_movement,
                   int end_vert_movement)
{
    _AX = 8;
    _CX = start_vert_movement;
    _DX = end_vert_movement;
    geninterrupt(0x33);
}

```

При перемещении манипулятора “мышь” по столу пропорционально движению перемещается и курсор “мыши”. Регулировать скорость перемещения курсора манипулятора по экрану позволяют пороги чувствительности, определяющие число “микки”, которое следует принять драйверу для перемещения курсора на один пиксел по горизонтали или вертикали. Малое значение чувствительности делает курсор “мыши” более подвижным. Например, при перемещении курсора по вертикали вниз с верхней линии до нижней в графическом режиме 10h (350 строк) и при чувствительности 1 микки/пиксел, для стандартной “мыши” 200 микки/дюйм потребуется переместить манипулятор на $350 / 200 * 25 = 44$ мм. Для перемещения курсора слева направо при этих же параметрах требуется перемещение “мыши” по столу на $640 / 200 * 25 = 80$ мм. Для установки чувствительности драйвера используется функция $AX = 000Fh$.

При вызове:

$AX = 000Fh$ — установка чувствительности драйвера “мыши” по горизонтали и вертикали;

CX — чувствительность драйвера по горизонтали в микки на 8 пикселей;

DX — чувствительность драйвера по вертикали в микки на 8 пикселей.

Возврат: нет.

Минимально возможное значение для CX и DX равно 1, что соответствует просто “реактивной” скорости. Даже случайные сотрясения на столе будут приводить к перемещению курсора “мыши”. Высокая чувствительность драйвера упрощает перемещение курсора на большие “расстояния” по экрану, но одновременно затрудняет установку курсора точно в заданную позицию на экране. Отмеченное противоречие между скоростью и точностью позиционирования разрешается драйвером мыши с помощью алгоритма баллистического курсора. Идея алгоритма заключается в следующем. Если “мышь” перемещается по столу с высокой скоростью, то информационные слова в адаптер поступают с высокой частотой. При достижении “мышью” так называемого порога удвоенной скорости драйвер начинает удваивать каждый принимаемый сигнал о перемещении. В результате вдвое возрастает скорость перемещения курсора манипулятора на экране. При уменьшении скорости перемещения манипулятора ниже упомянутого порога, удвоение прекращается и курсор по экрану движется со скоростью, определяемой установками чувствительности. Другими словами, для “далекого” перемещения курсора по экрану нужно сделать “мышью” по столу резкое движение в нужном направлении, а затем медленно подвести курсор в нужную точку. Установку собственного значения порога удвоенной скорости выполняет функция $AX = 0013h$.

При вызове:

$AX = 0013h$ — установка значения порога удвоенной скорости;

DX — порог удвоенной скорости курсора “мыши”, микки/с.

Возврат: нет.

По умолчанию порог удвоенной скорости курсора равен 64 микки/с.

Для определения текущих установок чувствительности и порога удвоенной скорости курсора “мыши” используется функция $AX=001Bh$.

При вызове:

$AX = 001Bh$ — определение текущих значений чувствительности по горизонтали и вертикали.

Возврат:

BX — чувствительность драйвера по горизонтали в микки на один пиксел;

CX — чувствительность драйвера по вертикали в микки на один пиксел;

DH — порог удвоенной скорости курсора “мыши” (количество микки в секунду, при достижении которого скорость перемещения курсора по экрану удваивается).

По умолчанию порог удвоенной скорости курсора равен 64 микки/с.

После проведенной инициализации программа должна “включить” курсор, что делает его видимым на экране. Эта операция деблокирует секцию обработчика прерываний от адаптера, работающую с видеопамятью. Сразу после инициализации курсор выключен. Состояние курсора отслеживает специальная внутренняя переменная, устанавливаемая после инициализации “мыши” в -1. Управление внутренним флагом осуществляют две функции прерывания 33h: **AX** = 0001, делающая курсор видимым, т.е. увеличивающая на 1 флаг видимости, и **AX** = 0002, которая делает курсор невидимым, т.е. уменьшает на 1 флаг видимости. Если флаг уже равен 0, его дальнейшего увеличения функцией **AX** = 0001 не происходит. Однако функция **AX** = 0002 уменьшает флаг, даже если он уже равен -1. В этой связи сразу после инициализации выполняется функция включения курсора и на каждую последующую операцию включения должна приходиться точно одна, а не более операций выключения. В противном случае курсор “мыши” не будет появляться на экране после его включения.

Все изменения информации на экране следует выполнять с выключенным курсором во избежание непредвиденных осложнений. Дело в том, что изменения, проведенные точно под курсором, не будут “известны” драйверу курсора и при перемещении в новую позицию экрана драйвер восстановит не новое, а прежнее значение видеопамати. После выполненных изменений на экране следует вновь включить курсор. Выключение курсора сбрасывает как форму, так и установку “горячего пятна” графического курсора.

Далее приводятся два макро для включения и выключения курсора:

```
#define CURSOR_ON()      _AX=1; geninterrupt(0x33);  
#define CURSOR_OFF()    _AX=2; geninterrupt(0x33);
```


4.3.2. Чтение позиции курсора и состояния кнопок “мыши”

После того, как выполнены все необходимые подготовительные операции, программа может использовать манипулятор для ввода информации. Вводимая информация — это текущая позиция курсора и состояние кнопок манипулятора. Периодический ввод информации с “мыши” и попадание курсора в те или иные области экрана позволяет выполнить нужные действия — “открыть” подменю, выполнить “заливку” той или иной области другим цветом и многое другое. Определение состояния кнопки (нажата, отпущена, нажата дважды) используется как команда на выполнение программой дополнительных действий. Де-факто сложился стандарт использования кнопок “мыши”. Например, нажатие левой кнопки для “правшей” означает выбор того или иного элемента меню либо включение какого-нибудь режима. Во многих прикладных программах нажатие левой кнопки аналогично нажатию клавиши ENTER. Быстрое двукратное нажатие левой или правой кнопки манипулятора в некоторых случаях используется как дополнительная команда на выполнение каких-либо действий программы, например запуск какой-либо программы на выполнение. Правая кнопка, напротив, отменяет сделанный ранее выбор, т. е. часто служит аналогом клавиши клавиатуры “Esc”.

Для определения текущего состояния манипулятора используются функции AX = 0003h; 0005h; 0006h; 000Bh.

При вызове: AX = 0003h — определение местоположения курсора и состояния кнопок манипулятора.

Возврат:

BL — байт состояния кнопок манипулятора:

бит 0 равен 1 — нажата левая кнопка;

равен 0 — не нажата левая кнопка;

бит 1 равен 1 — нажата правая кнопка;

равен 0 — не нажата правая кнопка;

бит 2 равен 1 — нажата средняя кнопка (для Mouse Systems);

равен 0 — не нажата средняя кнопка;

биты 3 - 7 — не используются;

CX — горизонтальная координата курсора “мыши” (номер столбца на экране);

DX — вертикальная координата курсора “мыши” (номер строки на экране).

В графическом режиме возвращаются пиксельные координаты, а в текстовом — “виртуальные” пиксельные координаты (см. описание функций $AX = 0007h$ и $AX = 0008h$ в п.4.3.1), и для получения номеров текстовой строки и столбца возвращаемые в CX и DX значения следует разделить на 8.

При вызове:

- $AX = 0005h$ — определение количества нажатий кнопок “мыши”;
- $AX = 0006h$ — определение количества отпусканий кнопок “мыши”;
- BX — идентификатор кнопки манипулятора:
 - 0 — запрос о левой кнопке;
 - 1 — запрос о правой кнопке;
 - 2 — запрос о средней кнопке (для Mouse Systems).

Возврат:

- BX — число нажатий (отпусканий) кнопки манипулятора с момента последнего обращения к функциям или с момента инициализации “мыши”, если выполняемый запрос — первый;
- CX — горизонтальная координата курсора “мыши” (номер столбца на экране) в момент нажатия запрошенной кнопки;
- DX — вертикальная координата курсора “мыши” (номер строки на экране) в момент нажатия запрошенной кнопки.

В графическом режиме возвращаются пиксельные координаты, в текстовом — “виртуальные” пиксельные координаты (см. описание функций $AX = 0007h$ и $AX = 0008h$ в п.4.3.1) и для получения номера текстовой строки и столбца возвращаемые в CX и DX значения следует разделить на 8. Счетчик числа нажатий (отпусканий) хранит значения от 0 до $FFFFh$. После вызова функций $AX = 0005h$ или $AX = 0006h$ соответствующий счетчик обнуляется.

При вызове: $AX = 000Bh$ — определение значения счетчиков сигналов микки.

Возврат:

- CX — перемещение курсора “мыши” по горизонтали в микки с момента последнего вызова данной функции или с момента инициализации “мыши”, если выполняемый запрос — первый;
- DX — перемещение курсора “мыши” по вертикали в микки с момента последнего вызова данной функции или с момента инициализации “мыши”, если выполняемый запрос — первый.

Возвращаемые в регистрах CX и DX значения имеют тип `int`. Отрицательные значения соответствуют движению курсора влево или вниз, положительные — вправо

или вверх. После каждого вызова функции значения счетчиков обнуляются. Для пересчета значений счетчиков в пиксели графического режима или в “виртуальные” пиксели текстового режима используется функция `AH = 001Bh` определения текущего значения чувствительности по горизонтали и вертикали.

Приведем текст Си-функции определения текущих координат курсора и состояния кнопок манипулятора “мышь”:

```
/* L4_5.C
```

```
Определяет состояние кнопок манипулятора “мышь” и
текущие координаты курсора. Помещает информацию в
структурную переменную по шаблону struct MOUSE_STATE.
Точка вызова описывает переменную и передает на нее
указатель. */
```

```
#include <dos.h>
```

```
struct MOUSE_STATE
```

```
{
    char buttons; /* байт состояния кнопок “мыши” */
    int hor_position; /* горизонтальная координата
                      курсора */
    int ver_position; /* вертикальная координата
                      курсора */
};
```

```
void ms_state(struct MOUSE_STATE * current_state)
```

```
{
    struct REGPACK r;
    r.r_ax = 3;
    intr(0x33, &r);
    current_state->buttons = r.r_bx;
    current_state->hor_position = r.r_cx;
    current_state->ver_position = r.r_dx;
}
```

Функция `ms_state()` возвращает состояние манипулятора на момент обращения к функции. В случае, когда программе необходимо зафиксировать одинарный или двойной “щелчок” (click) кнопки манипулятора (нажатие

и отпускание), удобнее пользоваться приводимой далее функцией, определяющей число отпусканий кнопки.

```
/* L4_6.C
```

```
Заполняет структурную переменную по шаблону MOUSE_STATE,
на которую указывает current_state, информацией о текущей
позиции курсора "мыши". Возвращает число отпусканий
кнопки. Поле buttons структурной переменной указывает
кнопку, о состоянии которой делается запрос. */
```

```
#include <dos.h>
```

```
struct MOUSE_STATE
```

```
{
    char buttons;      /* байт состояния/индикации
                        кнопок "мыши" */
    int hor_position; /* горизонтальная координата
                        курсора */
    int ver_position; /* вертикальная координата
                        курсора */
};
```

```
int ms_release(struct MOUSE_STATE * current_state)
```

```
{
    struct REGPACK r;
    r.r_ax = 6;
    r.r_bx = current_state -> buttons;
    intr(0x33, &r);
    current_state -> hor_position = r.r_cx;
    current_state -> ver_position = r.r_dx;
    return r.r_bx;
}
```

4.3.3. Позиционирование курсора "мыши" манипулятором и прикладной программой

При перемещении манипулятора по столу драйвер перемещает курсор "мыши" по экрану без какого-либо участия программы. Тем не менее и прикладная программа имеет возможность управлять позицией курсора "мыши". Для этого используется функция AX = 0004h.

При вызове:

AX = 0004h — установка курсора “мыши” и внутренних счетчиков позиции в новое место экрана;

CX — горизонтальная координата курсора “мыши”;

DX — вертикальная координата курсора “мыши”.

Возврат: нет.

Новые координаты задаются для графического режима в пикселах, а для текстового режима — в “виртуальных” пикселах (см. описание функций AX = 0007, AX = 0008 в п.4.3.1). Поэтому при необходимости установки курсора манипулятора в нужные текстовые строку и столбец их номера умножают на 8. Данная функция не пересекает границы, установленные функциями AX = 0007 и AX = 0008. При запросе позиционирования на точку экрана вне диапазона перемещения курсора последний упирается в ближайшую границу, но не пересекает ее.

Далее приводится Си-функция позиционирования курсора “мыши” в точку экрана с горизонтальной и вертикальной координатами `int hor_position`, `int ver_position`:

```
/* L4_7.C
```

```
Перемещает курсор “мыши” в заданную позицию экрана.
```

```
Координаты для текстового режима задаются в
```

```
“виртуальных” пикселах, т.е. номера текстовых строки и столбца умножаются на 8. */
```

```
#include <dos.h>
```

```
void ms_move(int hor_position, int ver_position)
```

```
{
```

```
    _AX = 4;
```

```
    _CX = hor_position;
```

```
    _DX = ver_position;
```

```
    geninterrupt(0x33);
```

```
}
```

Наличие функции управления позицией курсора “мыши” позволяет организовать управление единственным курсором на экране как самим манипулятором, так и с клавиатуры. В текстовом режиме такое совместное позиционирование курсора “мыши” и текстового аппаратного

курсора часто используется при построении различных текстовых редакторов. Функция приема клавиатурного ввода постоянно анализирует текущую позицию курсора “мыши” и приводит в соответствие с ней позицию текстового курсора BIOSa. При нажатии клавиш клавиатуры со стрелками (Left, Right, Up, Down и др.) выполняется позиционирование курсора “мыши”. Рассмотренные действия приводят к тому, что текущая позиция курсора, известная BIOSy и используемая для вывода информации на экран, управляется как манипулятором, так и клавишами со стрелками.

Ниже приводится пример демонстрационной программы, читающей состояние манипулятора “мышь” и отображающей все перемещения курсора “мыши” в пределах экрана в верхней строке в виде:

Строка:xx Столбец:xx Нажата xxxxxxxx кнопка манипулятора

При построении рассматриваемой программы используются функции вывода текстовой информации на экран, рассмотренные в гл. 8 и 9 второй книги комплекса: `hor_rpn()` — вывод строки символов горизонтально с заданным атрибутом, начиная с заданных строки и столбца (см. L9_9.C во второй книге комплекса); `scroll()` — вертикальный скроллинг окна экрана (см. L9_5.C во второй книге); `printche()` — вывод символа с заданным атрибутом заданное число раз, начиная с заданных строки и столбца (см. L9_8.C во второй книге); `goto_xy()` — позиционирование курсора с обновлением области данных BIOSa (см. L8_11.C во второй книге).

/* L4_8.C

Демонстрационная программа ввода информации с манипулятора “мышь”. Позиция курсора BIOSa периодически приводится в соответствие с текущей позицией курсора “мыши”. При движении курсора манипулятора на экране “рисуются” звездочка в текущей позиции курсора экрана.

*/


```

#include <dos.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "screen.h" /* см. 8.4 второй книги комплекса */

#define CURSOR_ON()      _AX=1; geninterrupt(0x33);
#define CURSOR_OFF()    _AX=2; geninterrupt(0x33);

struct MOUSE_STATE
{
    char buttons;      /*байт состояния кнопок "мыши"      */
    int hor_position; /*горизонтальная координата курсора*/
    int ver_position; /*вертикальная координата курсора */
};

/* Прототипы используемых функций. */
void set_text_cursor(int, unsigned int, unsigned int);
void ms_state(struct MOUSE_STATE *);
void ms_move(int, int);
void vert_movement(int, int);
void hor_movement(int, int);
void prn_repeat(char, char, int);          /* L3_13.C */

/* Внешние переменные,используемые функцией hor_prn().*/
unsigned vid_memory = 0xb800;
unsigned start_adr  = 0x0000;
unsigned max_stolb  = 80;
unsigned max_str    = 25;

/* Массив выводимых строк. */
char *strings[] = {" левая ", " правая", "средняя",
    "Строка xx; Столбец xx. Нахата xxxxxxxx кнопка",
    "xxxxxxxx" };

void main(void)
{
    struct MOUSE_STATE cur_state;
    char buf[3];

```



```

buf[2] = 0;
/* "Заливка" экрана скроллингом.
    Вывод строки состояния вверху экрана. */
scroll(ENTIRE, 0, 0, 24, 79, 0x17);
hor_prn(0, 2, strings[3], 0x1e);
printche(1, 0, '\xC4', 0x1e, 50); /* горизонтальная
                                   черта */

/* Инициализация "мыши". */
_AX=0;
geninterrupt(0x33);
if(_AX==0)
{
    puts("НЕТ \"мыши\" !");
    exit(-1);
}

/* Инициализация курсора "мыши" в текстовом режиме
    ("жесткий" курсор). */
set_text_cursor(1, 0x000e, 0x000f);
/* Установка вертикальных и горизонтальных
    границ курсора. */
hor_movement(0, 639);
vert_movement(16, 199);
/* Установка курсора в начальную позицию:
    2-я строка, 0-й столбец. */
ms_move(0, 16);
/* Включение курсора "мыши". */
CURSOR_ON();
/* Бесконечный цикл чтения ввода с клавиатуры.
    Выход - по нажатию двух кнопок одновременно. */
while(1)
{
    /* Определение текущей позиции курсора "мыши"
        и состояния кнопок. */
    ms_state(&cur_state);
    if(cur_state.buttons == 0x01) /* нажата
                                   левая кнопка */
        hor_prn(0, 32, strings[0], 0x1e);
    else

```



```

        if(cur_state.buttons == 0x02) /* нажата
                                         правая кнопка */
            hor_prn(0, 32, strings[1], 0x1e);
    else
        if(cur_state.buttons == 0x04)/* нажата
                                         средняя кнопка */
            hor_prn(0, 32, strings[2], 0x1e);
    else
        if(cur_state.buttons == 0x03) /* нажаты
                                         две кнопки */
        {
            /* Выключение курсора. */
            CURSOR_OFF();
            exit(0);
        }
    else
        if(cur_state.buttons == 0)/* нет
                                         нажатий кнопок */
            hor_prn(0, 32, strings[4], 0x1e);
    /*Перевод "виртуальных" координат в текстовые. */
    cur_state.hor_position >>= 3;
    cur_state.ver_position >>= 3;
    /* Обновление позиции курсора BIOS. */
    goto_xy(cur_state.ver_position,
            cur_state.hor_position, 0);
    /*Отображение номеров строки и столбца курсора.*/
    itoa(cur_state.ver_position, buf, 10);
    if(strlen(buf) == 1)
        strcat(buf, " ");
    hor_prn(0, 9, buf, 0x1e);
    itoa(cur_state.hor_position, buf, 10);
    if(strlen(buf) == 1)
        strcat(buf, " ");
    hor_prn(0, 21, buf, 0x1e);
    /*Демонстрация того, что перемещение курсора "мыши"
       сопровождается изменением текущей позиции BIOSa.
       При доступе к экрану курсор сначала выключается,
       а затем вновь включается. */
    CURSOR_OFF();

```



```

        prn_repeat('*', 0x1f, 1);
        CURSOR_ON();
    }
}

```

Для получения загрузочного модуля следует, используя файл проекта, выполнить совместную компиляцию файлов L3_13.C, L4_1.C, L4_3.C, L4_5.C — L4_8.C данной главы, а также L8_11.C, L9_5.C, L9_8.C, L9_9.C из второй книги комплекса.

Приводимая далее демонстрационная программа показывает, как можно использовать ввод с манипулятора “мышь” для “рисования”. В этой программе адаптер устанавливается в графический режим 12h. Далее описывается собственный графический курсор, имеющий форму, рассмотренную в п.4.3.1 (см. рис. 4.2). Как и в предыдущем примере, в верхней части экрана выводятся номера строки и столбца курсора манипулятора. Кроме того, отображается код цвета пиксела, используемый для “рисования”. Для вывода текста в графическом режиме используются варианты функций scroll(), hor_prn(), и printche(), работающие через BIOS (см. L9_5.C, L9_2.C и L9_3.C во второй книге комплекса; позиционирование курсора выполняет функция goto_xy() L8_11.C). При нажатой левой кнопке манипулятора, последний “рисует” пиксел в текущей позиции. При каждом нажатии правой кнопки код цвета рисуемого пиксела увеличивается на 1. При одновременном нажатии правой и левой кнопок демонстрационная программа завершает свою работу. Для вывода пиксела на экран используется функция plot(), рассмотренная в 10.3 второй книги комплекса (см. L10_5.C).

```
/* L4_9.C
```

```

Устанавливает в графическом режиме 12h курсор
манипулятора и затем отображает его перемещение в
строке состояния. При нажатой левой кнопке “рисует”
пиксел. При каждом нажатии правой кнопки код цвета

```


рисуемого пиксела увеличивается на 1. При одновременном нажатии правой и левой кнопок программа завершает свою работу. */

```
#include <dos.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "screen.h" /* см. 8.4 второй книги комплекса */
```

```
#define CURSOR_ON()      _AX=1; geninterrupt(0x33);
#define CURSOR_OFF()    _AX=2; geninterrupt(0x33);
#define RIGHT_BUTTON_QUERY      1
#define LEFT_BUTTON_DOWN        1
#define LEFT_AND_RIGHT_BUTTON_DOWN 3
```

```
struct MOUSE_STATE
{
    char buttons;      /* байт состояния/индикации
                        кнопок "мыши"*/
    int hor_position; /* горизонтальная
                        координата курсора */
    int ver_position; /* вертикальная
                        координата курсора */
};
```

/* Прототипы используемых функций. */

```
void set_text_cursor(int, unsigned int, unsigned int);
void ms_state(struct MOUSE_STATE *);
void ms_move(int, int);
void vert_movement(int, int);
void hor_movement(int, int);
void set_graph_cursor(int, int, unsigned int far *);
int ms_release(struct MOUSE_STATE *);
```

```
/* Внешние переменные, используемые функцией plot(). */
int put_mode = 0; /* режим вывода пиксела - COPY */
int page = 0; /* номер видеостраницы */
int bytes_per_line = 80; /* число адресов
```



```

                                на одну ТВ-строку */
int bytes_per_page = 0x8000; /* число адресов
                                на одну страницу*/
int max_page = 0; /* максимально возможный
                                номер страницы */
int max_x = 639; /* число пикселей в строке - 1 */
int max_y = 479; /* число строк на экране - 1 */
/* Массив строк, выводимых вверху экрана. */
char *strings[] =
{
    "Row xxx; Column xxx; Color xx.",
    "Left button - drawing,"\
    "Right button - color changing,"\
    "Left & Right - termination"
};

void main(void)
{
    struct MOUSE_STATE cur_state, work;
    int old_color = 0xf; /* текущий цвет для
                            "рисования" пиксела */
    int new_color = 0xf;
    int old_ver_position, old_hor_position;
    static unsigned screen_and_cursor_masks[] =
    {
        /* Видимый на любом фоне графический курсор.
           AND-маска. */
        0xffff, 0xffff, 0xffff, 0xffff,
        0xffff, 0xffff, 0xffff, 0xffff,
        0xffff, 0xffff, 0xffff, 0xffff,
        0xffff, 0xffff, 0xffff, 0xffff,
        /* XOR-маска. */
        0x8000, 0xC000, 0xA000, 0x9000,
        0xA800, 0xB400, 0xBA00, 0xBD00,
        0xBE80, 0xBF40, 0xBFA0, 0xBF00,
        0xBF00, 0xAE00, 0xA300, 0xE300,
    };
    char buf[3];
    buf[2] = 0;

```



```

/* Переключение в графический режим 12h. */
_AX = 0x0012;
geninterrupt(0x10);
/* Инициализация "мыши". */
_AX=0;
geninterrupt(0x33);
if(_AX==0)
{
    puts("НЕТ \"мыши\" !!!");
    exit(-1);
}
/* "Заливка" экрана скроллингом.
Вывод строки состояния вверху экрана. */
scroll(ENTIRE, 0, 0, 29, 79, 0x0);
hor_prn(0, 2, strings[0], 0xf);
printche(1, 0, '', 0x9, 30);
hor_prn(2, 0, strings[1], 0xf);
itoa(old_color, buf, 10);
if(strlen(buf) == 1)
    strcat(buf, " ");
hor_prn(0, 29, buf, 0xf);
/* Инициализация курсора "мыши" в графическом режиме.
"Горячее пятно" - левый верхний
угол прямоугольника курсора. */
set_graph_cursor(0, 0, screen_and_cursor_masks);
/* Установка вертикальных и горизонтальных границ
курсора. Задаются пиксельные координаты.
Знакоместо - 8 x 16. */
hor_movement(0, 639);
vert_movement(48, 479);
/* Установка курсора в начальную позицию:
3-я строка, 0-й столбец. */
ms_move(0, 48);
old_ver_position = 0;
old_hor_position = 48;
/* Включение курсора "мыши". */
CURSOR_ON();
/* Бесконечный цикл чтения ввода
с манипулятора "мышь". */

```



```

while(1)
{
    /* Определение текущих позиции курсора "мыши"
       и состояния кнопок. Выход по нажатии двух
       кнопок одновременно. */
    ms_state(&cur_state);
    switch(cur_state.buttons)
    {
case LEFT_BUTTON_DOWN: /* нажата левая кнопка */
    /* "Рисование" пиксела, но при условии, что
       координаты курсора изменились. */
    if(old_ver_position != cur_state.ver_position |
       old_hor_position != cur_state.hor_position)
    {
        CURSOR_OFF();
        plot(cur_state.hor_position,
              cur_state.ver_position, old_color);
        CURSOR_ON();
    }
    break;
case LEFT_AND_RIGHT_BUTTON_DOWN: /* нажаты две кнопки */
    CURSOR_OFF();
    /* Восстановление текстового режима 3. */
    _AX = 0x0003;
    geninterrupt(0x10);
    exit(0); /* завершение программы */
    break;
    }
    /* Обработка ситуации изменения цвета пиксела. */
    work.buttons = RIGHT_BUTTON_QUERY;
    if(ms_release(&work)) /* правая кнопка отпущена ? */
        new_color = (old_color + 1) & 0x000f;
    /* Отображение номера строки, столбца курсора и
       цвета. Выполняется только в том случае, если
       позиция курсора манипулятора изменилась. */
    if(old_ver_position != cur_state.ver_position)
    {
        old_ver_position = cur_state.ver_position;
        itoa(cur_state.ver_position, buf, 10);
    }
}

```



```

        if(strlen(buf) == 2)/* добавление пробелов */
            strcat(buf, " ");/* до строки из 3-х цифр */
        else
            if(strlen(buf) == 1)
                strcat(buf, " ");
            hor_prn(0, 6, buf, 0xf);
    }
    if(old_hor_position != cur_state.hor_position)
    {
        old_hor_position = cur_state.hor_position;
        itoa(cur_state.hor_position, buf, 10);
        if(strlen(buf) == 2) /* добавление пробелов */
            strcat(buf, " "); /* до строки из 3-х цифр */
        else
            if(strlen(buf) == 1)
                strcat(buf, " ");
            hor_prn(0, 18, buf, 0xf);
    }
    if(old_color != new_color)
    {
        itoa(new_color, buf, 10);
        if(strlen(buf) == 1) /* добавление пробела */
            strcat(buf, " ");/* до строки из 2-х цифр */
        hor_prn(0, 29, buf, 0xf);
        old_color = new_color;
    }
} /* граница бесконечного цикла */
}

```

Для получения загрузочного модуля следует, используя файл проекта, выполнить совместную компиляцию файлов L4_1.C — L4_5.C, L4_7.C — L4_9.C данной главы и файлов L8_11.C, L9_2.C, L9_3.C и L9_5.C, рассмотренных во второй книге комплекса.

Интерфейс прикладной программы с драйвером “мыши” включает и многие другие функции, предназначенные для установки собственных программ обработки прерываний от манипулятора, получения дополнительной информации об аппаратуре манипулятора и иных действиях.

5. УПРАВЛЕНИЕ АДАПТЕРОМ ПАРАЛЛЕЛЬНОЙ СВЯЗИ И ПРИНТЕРОМ

5.1. Технические и программные средства управления адаптером параллельной связи и принтером

Для подключения периферийных устройств к персональному компьютеру IBM PC используется не только последовательный, но и параллельный интерфейс. Управление передачей информации по параллельному интерфейсу выполняет специальная электронная схема, называемая *адаптером параллельной связи* или параллельным адаптером. Архитектура персонального компьютера типа IBM PC допускает подключение до трех устройств по параллельному интерфейсу. Ссылка на такие устройства (и соответствующие им адаптеры) осуществляется по символическим именам LPT1 — LPT3. В данной главе рассматривается управление адаптерами параллельной связи, к которым подключаются принтеры по параллельному интерфейсу. Часто даже в технической документации системные программные средства управления адаптерами параллельной связи называют средствами управления принтером. В данной главе мы будем по возможности разделять два элемента: адаптер и собственно принтер как периферийное устройство.

Параллельные принтеры подключаются к компьютеру по однонаправленному параллельному асинхронному интерфейсу. Асинхронность означает, что каждый очередной байт передается из адаптера параллельной связи компьютера только после получения сигнала от принтера, подтверждающего прием предыдущего байта. Однонаправленность означает, что байты данных передаются только от процессора к принтеру. Принтер имеет несколько обратных линий, используемых для сигнализации своего состояния. Асинхронность интерфейса чревата угрозой “повисания” компьютера. Например, если принтер неисправен, то процессор, ожидающий сигнала подтверждения на переданный байт, никогда его не дожидется. Защитой от “повисания” является ограничение времени ожидания ответа принтера некоторым числом секунд. По истечении этого промежутка времени фиксируется ошибка. Такой метод называется *контролем по тайм-ауту* и реализуется либо самой прикладной программой, управляющей принтером через порты адаптера параллельной связи, либо автоматически функциями BIOSa.

Сам принтер представляет собой сложное электромеханическое устройство с собственной микропроцессорной системой управления, памятью, называемой *буфером печати*, и постоянным запоминающим устройством (ПЗУ), хранящим программы системы управления принтером. Принтер способен принимать и накапливать байты данных в буфере и одновременно печатать ранее полученные байты. Физически буфер печати организуется как кольцевой со своими указателями “головы” и “хвоста” (в этом смысле он очень похож на буфер клавиатуры, рассмотренный подробно в п.7.2.3 второй книги). Когда байт данных записан в буфер печати, процессор считает его уже “отпечатанным”, хотя сам момент печати данного символа может наступить значительно позже. Когда буфер принтера наполнится, последний прекращает прием байтов данных. Современные матричные принтеры имеют буфер печати от 8 до 32К байт, т.е. способны накапливать несколько страниц текста. Наиболее совершенные лазерные принтеры могут иметь буфер размером в не-

сколько мегабайтов. Совмещение приема данных с их печатью и накопление данных в буфере порождают ситуацию, когда процессор уже передал весь файл и продолжает свою работу, а принтер еще достаточно долго печатает файл. Принтер способен распознавать и выполнять ряд приказов, передаваемых центральным процессором как обычные байты данных. К сожалению, коды и формат приказов зависят от типа принтера. Как правило, приказами являются символы с кодами от 0 до 31, а также специальные последовательности символов, начинающиеся с байта с кодом 27 в десятичной системе счисления (символ Esc). Специальный управляющий символ (для принтеров типа Epson его код равен 24 в десятичной системе счисления) вызывает переход принтера в состояние по умолчанию. *Состояние по умолчанию* — это состояние, в которое попадает принтер сразу после включения электропитания. Оно характеризуется:

- 1) выбранным режимом работы принтера (текстовый или графический);
- 2) выбранным шрифтом (принтеры часто имеют в ПЗУ целый набор шрифтов печати);
- 3) интервалом между строками;
- 4) числом строк на одной форме (обычно, листе бумаги);
- 5) интервалом между строками;
- 6) некоторыми другими зависящими от типа принтера параметрами.

5.2. Информация BIOSa о принтерах и других периферийных устройствах

BIOS содержит несколько областей данных, относящихся к адаптеру параллельной связи. Прежде всего, это слово по адресу 40:10h, называемое *словом подключенного оборудования*. Формат слова приведен в табл. 5.1.

Табл. 5.1. Информация BIOSa о подключенном оборудовании
(слово по адресу 40:10h)

Биты	О п и с а н и е
15 - 14	Число адаптеров параллельных устройств (принтеров)
13 - 12	Зарезервированы
11 - 9	Число адаптеров последовательной связи
8	Зарезервирован
7 - 6	Число подключенных накопителей на гибких магнитных дисках(НГМД), если бит 0 равен 1: 00 - 1 НГМД; 01 - 2 НГМД; 10 - 3 НГМД; 11 - 4 НГМД
5 - 4	Устанавливаемый при загрузке системы режим видеоадаптера дисплея: 00 - зарезервировано; 01 - текстовый 40 x 25 цветной (режим 1); 10 - текстовый 80 x 25 цветной (режим 3); 11 - текстовый 80 x 25 монохроматический (режим 7)
3	Зарезервирован
2	Наличие позиционирующего устройства
1	Наличие сопроцессора математики с плавающей точкой
0	Указатель наличия НГМД (см. описание битов 7 и 6)

П р и м е ч а н и е. Некоторые персональные компьютеры используют биты, помеченные как зарезервированные, для сообщения некоторой дополнительной информации.

Та же самая информация может быть получена программой и через прерывание 11h BIOS.

Отметим, что слово подключенного оборудования заполняется BIOSом во время начальной загрузки системы по результатам тестирования внутренних узлов компьютера и в этой связи может и не отображать корректно текущее состояние оборудования. Что же касается принтеров, то рассматриваемое слово сообщает не число физически подключенных к компьютеру параллельных принтеров, а только число адаптеров параллельной связи. Кроме того, не учитываются принтеры, которые могли бы быть подключены к компьютеру по последовательному интерфейсу. В случае, когда требуется абсолютно точно узнать число физически подключенных к компьютеру принтеров, приходится спрашивать об этом пользователя.

Следующая область данных о принтере — это три байта по адресу 40:78h. Здесь располагается по одному байту для каждого из трех возможных адаптеров парал-

лельной связи. Байт хранит значение времени тайм-аута в секундах. Это значение используется как счетчик BIOS-функциями прерывания 17h при выполнении передачи данных. Так как значение интервала тайм-аута сначала уменьшается на 1, а затем используется как установка в счетчик, значение 0 соответствует на самом деле не нулевому интервалу, а интервалу в 255 секунд.

В области данных BIOSa, начиная с адреса 40:08h, хранятся три слова так называемых *базовых адресов (БА)* адаптеров: по адресу 40:08h располагается базовый адрес LPT1 (БА1), по адресу 40:0Ah — LPT2 (БА2), по адресу 40:0Ch — LPT3 (БА3). Если в слове записан 0, это говорит об отсутствии соответствующего адаптера параллельной связи.

5.3. Порты адаптера параллельной связи

Каждый из адаптеров параллельной связи имеет три порта: порт данных, номер которого совпадает с базовым адресом адаптера (БА+0), порт состояния (БА+1) и порт управления (БА+2).

В *порт данных* записывается байт, посылаемый принтеру. Следует иметь в виду, что простая запись байта в порт данных не означает передачу информации через интерфейс. Передача байта должна инициироваться записью необходимых байтов в *порт управления*. Значения отдельных битов порта управления приведены в табл. 5.2.

Порт состояния делает доступными центральному процессору сигналы по обратным линиям интерфейса, идущим от принтера. Значения отдельных битов порта состояния приведены в табл. 5.3.

Обычная последовательность действий при передаче данных принтеру состоит из следующих шагов:

- 1) определяется число адаптеров параллельной связи и, если их несколько, выбирается один из них;
- 2) инициализируется адаптер параллельной связи;

Табл. 5.2. Биты порта управления адаптера параллельной связи (БА+2)

Биты	Функция
0	Запись 1 инициирует передачу байта по интерфейсу
1	Запись 1 вызывает автоматический перевод строки после возврата каретки
2	Запись 0 вызывает инициализацию адаптера параллельной связи и принтера
3	Запись 1 выбирает принтер, 0 отменяет выбор принтера
4	Запись 1 разрешает прерывания от адаптера параллельной связи, 0 запрещает прерывания
5-7	Не используются

Табл. 5.3. Биты порта состояния адаптера параллельной связи (БА+1)

Биты	Функция
0-2	Не используются
3	=0 - сигнал от принтера "Ошибка принтера"; =1 - нет сигнала от принтера "Ошибка принтера"
4	=0 - принтер в состоянии "off-line" (не подключен к линиям интерфейса); =1 - принтер в состоянии "on-line" (подключен к линиям интерфейса)
5	=0 - нет сигнала "Отсутствие бумаги"; =1 - сигнал от принтера "Отсутствие бумаги"
6	=0 - сигнал от принтера "Подтверждение приема символа"; =1 - нет сигнала от принтера "Подтверждение приема символа"
7	=0 - сигнал от принтера "Занято"; =1 - нет сигнала от принтера "Занято" (принтер свободен)

3) проверяется, подключен ли принтер к интерфейсу (находится ли принтер в состоянии on-line или off-line);

4) если принтер подключен, передается символ;

5) после каждого переданного символа анализируется порт состояния: если получен сигнал подтверждения, повторяется шаг 4; если сообщается та или иная ошибка (нет бумаги, ошибка принтера), выполняется процедура обработки ошибки.

5.4. Средства BIOSa управления адаптером параллельной связи

BIOS имеет в своем составе три функции прерывания 17h, используемые для управления принтером. Во всех функциях значение в регистре DX задает номер адаптера параллельной связи: 0 соответствует LPT1, 1 — LPT2, 2 — LPT3. При возвращении из функции регистр AH содержит байт состояния, несколько отличающийся от байта порта состояния (табл. 5.4).

Табл. 5.4. Байт состояния адаптера, сообщаемый функциями прерывания 17h

Биты	Функция
7	=0 - сигнал от принтера "Занято"; =1 - нет сигнала от принтера "Занято" (принтер свободен)
6	=0 - нет сигнала от принтера "Подтверждение приема символа"; =1 - сигнал от принтера "Подтверждение приема символа"
5	=0 - нет сигнала "Отсутствие бумаги"; =1 - сигнал от принтера "Отсутствие бумаги"
4	=0 - нет сигнала "Принтер выбран"; =1 - сигнал "Принтер выбран"
3	=1 - сигнал от принтера "Ошибка принтера"; =0 - нет сигнала от принтера "Ошибка принтера"
2 - 1	Не используются
0	=1 - наступило условие тайм-аута (исчерпано время ожидания какой-либо реакции принтера); =0 - условие тайм-аута не наступило

Функция AH = 0 передает на печать символ, помещенный в AL, AH = 1 инициализирует адаптер параллельной связи, а AH = 2 читает порт состояния адаптера.

Каждая из функций прерывания 17h может быть легко реализована физическим доступом к портам адаптера параллельной связи. Но использование BIOSa для управле-

ния принтером наиболее целесообразно: во-первых, это повышает мобильность программы; во-вторых, сохраняет выполненную в системе “русификацию” принтера резидентными программами, например ZTKNEW.COM (Бордич), ВЕТТА.EXE (Чижов) и т.п.; в-третьих, обеспечивает автоматический контроль времени ожидания реакции принтера, что гарантирует отсутствие “повисания” программы, ожидающей ответа от выключенного внешнего устройства и др. Интервал ожидания легко может быть изменен записью другого значения по адресу 40:78h, 40:79h или 40:7Ah в зависимости от номера адаптера параллельной связи.

Непосредственный доступ к портам адаптера параллельной связи имеет смысл использовать в программах в тех редких случаях, когда “перехват” вывода на принтер нежелателен, например с целью ограничения несанкционированного распространения информации в защищенных от копирования справочниках или обучающих программах.

5.5. Инициализация адаптера параллельной связи и принтера

Программа, управляющая принтером, должна начинать свою работу с определения числа адаптеров параллельной связи, установленных в компьютере. Для этого используется информация BIOSa (см. 5.2). Далее приводится текст Си-функции, возвращающей число адаптеров параллельной связи, установленных в компьютере.

```
/* L5_1.C
```

```
Возвращает число установленных в системе адаптеров  
параллельной связи. */
```

```
int prt_inst(void)
```

```
{
```

```
    register unsigned int _es * ptr =
```

```
                                (unsigned int _es *) 0x10;
```

```
    _ES = 0x40;
```



```

        return((*ptr & 0xC000) >> 14); /* возвращает биты
                                   15, 14 слова по адресу 40:10h */
    }

```

Непосредственное управление принтером позволяет скорректировать значение интервала тайм-аута. Для этого новое значение интервала записывается в одно из слов области данных BIOSa. Далее приводится текст Си-функции, предназначенной для исполнения рассмотренной операции.

```

/*
L5_2.C
Устанавливает новое значение timeout_seconds
для параллельного принтера 0, 1 или 2.
Возврат:
    OK (0) - в случае успеха;
    BAD_PARAM (!= 0) - если заданы недопустимые
                        параметры. */

#define OK          0
#define BAD_PARAM  -1

int prt_inst(void); /* см. L5_1.C */
int prt_time_out(unsigned int timeout_seconds,
                  int printer)
{ register char _es * ptr = (char _es *) 0x78;
  if(printer > prt_inst())
    return BAD_PARAM;
  _ES = 0x40;
  ptr += printer << 1; /* смещение до нужного
                        байта памяти */
  * ptr = ++ timeout_seconds;
  return OK;
}

```

После того, как установлено наличие одного или более адаптеров параллельной связи, выбирается один из них и выполняется его инициализация. Инициализация принтера приводит к очистке буфера печати, установке принтера “Начало формы” и сбросу всех управляющих установок, полученных принтером ранее через интерфейс. В результате

принтер переводится в состояние по умолчанию, в котором он находился после включения электропитания. Инициализация может быть выполнена средствами BIOSa или непосредственным доступом к портам. Далее приводится пример Си-функции, инициализирующей принтер, подключенный к адаптеру параллельной связи printer. Функция использует BIOS-функцию AH = 01.

```
/*
L5_3.C
Инициализирует адаптер параллельной связи для
параллельного принтера 0, 1 или 2, ассоциируемого
соответственно с LPT1, LPT2 или LPT3. Работает через BIOS.
Возврат:
    OK (0)          - в случае успеха;
    BAD_PARAM (-1) - если заданы недопустимые параметры;
    > 0             - байт состояния BIOS-функции. */
#include <dos.h>
#define OK          0
#define BAD_PARAM  -1
int prt_inst(void); /* см. L5_1.C */
int prt_init(int printer)
{
    if(printer > prt_inst()) /* допустимый номер */
        return BAD_PARAM    /* принтера ?      */
    _AH = 0x01;
    _DX = printer;
    geninterrupt(0x17);
    if(_AH == 0x10) /* если инициализация успешна, */
        return OK; /* все биты, кроме бита 4, равны 0 */
    else
        return (_AH);
}
```

Перед посылкой символа на принтер программа должна проверить состояние принтера для того, чтобы избежать ситуации его неготовности к приему информации. Приведенная далее Си-функция использует функцию AH = 02 прерывания 17h BIOSa для определения состояния принтера и возврата значений битов 4 и 7. Эти биты будут единственными установленными битами, если принтер готов к приему информации. Другими словами, принтер

сообщает о том, что он “не занят” и “выбран”. Это — наиболее предпочтительный способ проверки, так как кодирование битов порта состояния зависит от типа принтера. Если рассматриваемая функция возвращает 0, принтер готов к приему информации. В противном случае принтер находится в состоянии off-line.

```
/*
L5_4.C
Анализирует состояние подключенного к адаптеру
параллельной связи параллельного принтера 0, 1
или 2, ассоциируемого соответственно с LPT1,
LPT2 или LPT3. Работает через BIOS.
Возврат:
    OK (0)          – принтер готов к приему информации;
    BAD_PARAM (-1) – если заданы недопустимые параметры;
    > 0             – байт состояния BIOS-функции. */
#include <dos.h>
#define OK          0
#define BAD_PARAM  -1
int prt_inst(void); /* см. L5_1.C */
int prt_ready(int printer)
{ if(printer > prt_inst()) /* допустимый          */
    return BAD_PARAM;     /* номер принтера ? */
    _AH = 0x02;
    _DX = printer;
    geninterrupt(0x17);
    _BH = _AH;             /* отгрузка возвращенного значения */
    if(_AH == 0x90) /* если принтер готов к приему, все */
        return OK;   /* биты, кроме битов 4 и 7, равны 0 */
    else
        return (_BH); /* регистр AH уже разрушен ! */
}
```

5.6. Передача символов

Передача символов на принтер может быть выполнена на уровне MS-DOS, на уровне BIOSa и непосредственно через порты принтера.

Доступ к принтеру на уровне MS-DOS выполняется через вывод данных в специальный символьный файл PRN (LPT1 — LPT3) (подробнее об этом см. в гл. 2 и 3 второй книги комплекса). Для вывода на принтер могут использоваться и функции библиотеки Turbo C: потоковые функции `fwrite()`, `fprintf()`, `fputs()` и др., использующие автоматически открытый поток `stdprn`, и префиксные функции `write()` и `_write()` вывода информации в файл с префиксом 4. Например, приведенный далее фрагмент кода выведет на печать строку, активизирует звуковой сигнал и затем прогонит бумагу до конца листа ("переведет" форму).

```
#include <stdio.h>
#include <io.h>
#include <string.h>
. . .
char * ptr = "ТЕСТОВАЯ СТРОКА\nЗВУКОВОЙ СИГНАЛ\a\xC\n";
write(4, ptr, strlen(ptr));
. . .
```

Использование средств MS-DOS для вывода данных на принтер имеет следующие достоинства:

1) для выполнения вывода MS-DOS "пропускает" его через драйвер принтера, что позволяет использовать для вывода на печать собственные устанавливаемые драйверы печати;

2) используются стандартные средства MS-DOS обработки ошибок, например выдача сообщения об ошибке и приглашение выбрать дальнейшие действия при неготовности принтера к работе:

```
Write fault error writing device PRN
Abort, Retry, Ignore, Fail
```

3) использование функций стандартного файлового вывода обеспечивает максимальную мобильность кода.

Доступ к файлам подробно рассмотрен в гл. 3 второй книги комплекса. Далее рассматривается вывод данных на печать на уровне BIOSa.

Передаче символов должна предшествовать инициализация адаптера и принтера, проверка готовности принтера к приему информации. Ранее рассмотренные функции `prt_init()` и `prt_ready()` выполняют эти операции. После того, как символ передан в принтер, следующий символ должен посылаться в принтер после получения сигнала подтверждения приема. Для этого обычно анализируют бит 6 кода возврата из BIOS-функции. В качестве примера далее приводится функция печати ASCIIZ-строки символов. Функция автоматически инициализирует принтер при первом обращении к ней. В случае возникновения ошибки функция возвращает байт состояния BIOSa и номер позиции следующего печатаемого символа. Предполагается, что ошибку принтера обрабатывает другая функция и после устранения причины ошибки печать строки может быть возобновлена с еще не отпечатанного символа.

/*

L5_5.C

Печатает ASCIIZ-строку символов `string` на принтере, ассоциируемом соответственно с LPT1, LPT2 или LPT3. Работает через BIOS.

Возврат:

OK (0) – информация передана в принтер;

BAD_PARAM (-1) – если заданы недопустимые параметры;

> 0 – байт состояния BIOS-функции.

`cur_position`, имеющая смысл только в случае возврата значения, большего 0, и сообщающая позицию следующего символа строки, с которой должна возобновляться печать после обработки ошибки. */

#include <dos.h>

#define OK 0

#define BAD_PARAM -1

int prt_inst(void); /* см. L5_1.C */

int prt_str(int printer, char * string,
 unsigned *cur_position)

{ register ret;

if(printer > prt_inst()) /* допустимый */

/* номер принтера ? */


```

    {
        * cur_position = 0;
        return BAD_PARAM;
    }
    /* Инициализация принтера. */
    if(ret = prt_init(printer))
    {
        * cur_position = 0;
        return BAD_PARAM;
    }
    while(*string)
    {
        _AL = * string;
        _AH = 0x01;
        _DX = printer;
        geninterrupt(0x17);
        _BH = _AH; /* отгрузка возвращенного значения */
        if(!(_AH == 0xd0)) /* если принтер принял преды-
                               дущий символ и готов к приему
                               следующего символа, все биты,
                               кроме битов 7, 6 и 4, равны 0*/
            return (_BH); /* регистр AH уже разрушен ! */
        (* cur_position)++;
        string++;
    }
}

```

В данном разделе рассмотрены лишь некоторые из вопросов по управлению адаптером параллельной связи. В принципе возможно использование адаптера параллельной связи не только для вывода информации, но и для ее ввода. Это позволяет с использованием специально изготовленных проводников и соответствующего программного обеспечения организовать перенос информации между компьютерами по значительно более быстрому параллельному интерфейсу. К сожалению, подробное рассмотрение этой возможности выходит за рамки данной книги.

6. УПРАВЛЕНИЕ ОТОБРАЖАЕМОЙ И РАСШИРЕННОЙ ПАМЯТЬЮ

6.1. Отображаемая память EMS

6.1.1. Элементы механизма отображаемой памяти

Отображаемая память (Expanded Memory) реализуется специальной аппаратной платой. Для построения таких плат и организации интерфейса прикладной программы по управлению отображаемой памятью выработаны специальные стандарты (спецификации), называемые *EMS (Expanded Memory Specification)*. Наиболее известным из них является стандарт, предложенный совместно компаниями Lotus, Intel и Microsoft, — спецификация LIM EMS. Спецификации постоянно расширяются и совершенствуются. Наиболее ранним является EMS версии 3.2 (опубликован в сентябре 1985 г.). В августе 1987г. был опубликован стандарт EMS версии 4.0, который и рассматривается в данной главе.

Идея EMS-памяти состоит в следующем. Адресное пространство в 1М байт распределяется в IBM PC при работе под управлением MS-DOS так, как показано на рис. 6.1.

Таким образом, прикладные программы могут использовать не более 640К байт оперативной памяти. Однако, если в системе установлена EMS-плата, появляется возможность использовать до 32М байт дополнительной памяти платы (так называемой *EMS-памяти*). Область адресов, зарезервированная для BIOSa, занята полностью

программами ПЗУ только в PS/2, а для IBM PC XT и IBM PC AT в этом диапазоне есть окно размером не менее 64К байт. Его и закрывает EMS-плата, реализуя механизм виртуального отображения страниц (рис. 6.2). Начальная граница окна (сегмент адреса) задается регистрами-конфигураторами платы, а в некоторых реализациях EMS может устанавливаться прикладной программой.

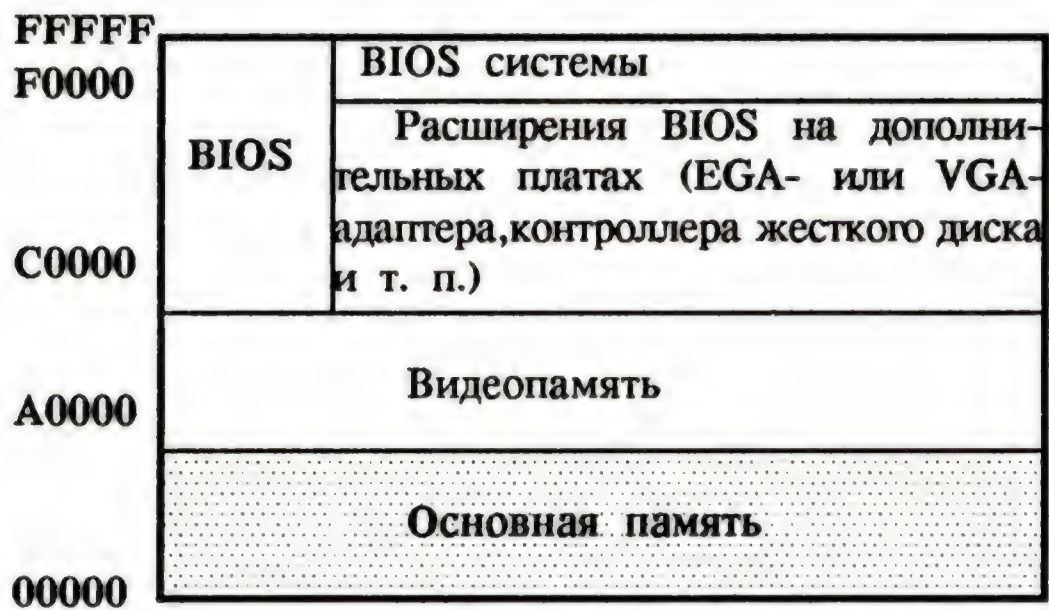


Рис. 6.1. Распределение адресов памяти в IBM PC

Отображаемая память имеет до 8М байт (EMS версии 3.2) или до 32М байт (EMS версии 4.0 и старше) на одной или нескольких платах адаптера отображаемой памяти. Совокупность четырех страниц памяти в адресном пространстве 1М байт называют *фреймом страниц*. Эти 64К байта памяти принадлежат EMS-плате. Запрос чтения или записи к адресам фрейма страниц аппаратно переадресовывается на логическую страницу. Закрепление физических страниц за логическими называют *картой отображения*. Карта может изменяться программно. Физические страницы имеют номера 0, 1, 2 и 3. Любая из четырех физических страниц может быть отображена на любую из логических страниц.

Логические страницы перед использованием должны быть распределены. В ходе операции распределения EMS-

памяти программа запрашивает нужное ей число логических страниц. В дальнейшем для ссылки на выделенные страницы используется целое число, называемое *ЕММ-префиксом* (*ЕММ-handle*). Логические страницы, закрепленные за префиксом, нумеруются от 0 до N-1, где N — число страниц, закрепленных операцией распределения за данным префиксом. Другими словами, логическая страница в группе закрепленных за префиксом страниц имеет относительный номер, а не абсолютный.

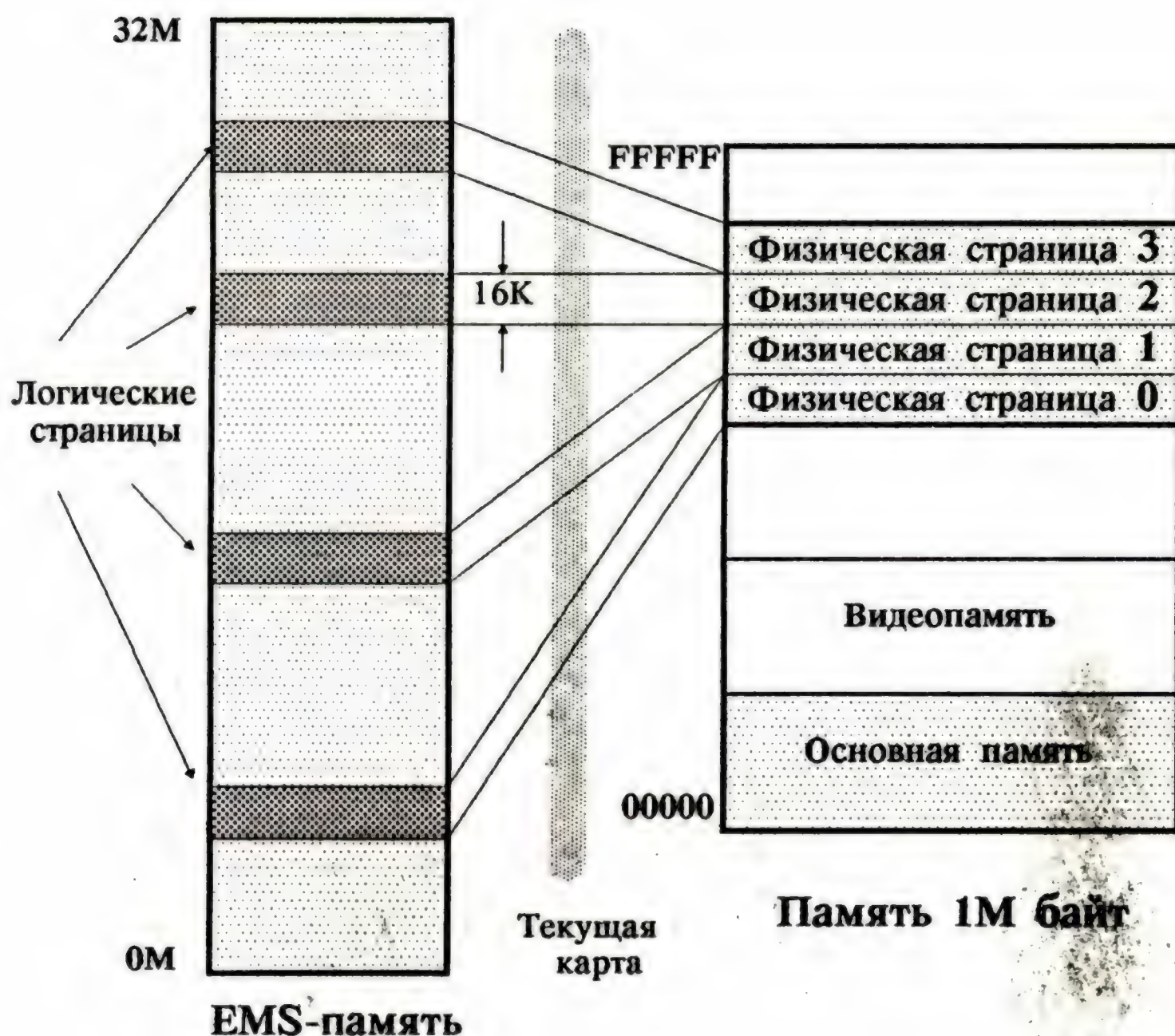


Рис. 6.2. Элементы механизма отображаемой (expanded) памяти

6.1.2. Основные функции интерфейса с отображаемой памятью

EMS-плата управляется рядом внутренних портов, доступ к которым осуществляют специальные драйверы отображаемой памяти, например, EMM.SYS, XMS2EMS.SYS и др. Далее такие драйверы будем называть *ЕММ-драйверами*. Для взаимодействия прикладной программы с ЕММ-драйвером используются функции прерывания 67h.

Прежде чем программа начнет использовать любую функцию EMS, необходимо определить наличие ЕММ-драйвера в системе. Если драйвер в ходе начальной загрузки системы был инсталлирован, выдача прерывания 67h будет безопасной. В противном случае вероятна ситуация обращения к прерыванию 67h, для которого отсутствует вектор в таблице векторов прерываний, и, как результат, — “повисание” системы. Один из двух рекомендуемых LIM EMS способов определения типа драйвера, обслуживающего отображаемую память, состоит в следующем. По EMS версии 3.2 вектор прерывания 67h после установки ЕММ-драйвера должен равняться указателю на заголовок инсталлируемого драйвера. Так как драйверы устанавливаются в памяти с выравниванием на границе параграфа, поле смещения в векторе 67h будет равно нулю. В заголовке драйвера, как это определено технической документацией MS-DOS, по смещению 0Ah располагается ASCIIZ-строка имени драйвера (не более 8 символов). Установленное для EMS имя должно быть “ЕММХХХХ\0”. Рассмотренный способ положен в основу приводимой далее функции EmsInstalled(), возвращающей символическую константу NOTINSTALLED (1), если ЕММ-драйвер не обнаружен. В противном случае функция возвращает символическую константу NOERROR (0).

/*

L6_1.C

Определяет наличие драйвера, управляющего EMS-памятью.

Возврат:

NOTINSTALLED (1) – если не инсталлирован ЕММ-драйвер;


```

NOERROR (0)    – если обнаружен драйвер EMS-памяти. */

#define    NOTINSTALLED    1
#define    NOERROR          0
#include <dos.h>

int EmsInstalled(void)
{
    static char * EmmName = "EMMXXXX0";
    char far * EmmPtr;
    /* Формирование указателя
    на имя драйвера в заголовке. */
    _AX = 0x3567; /* определение вектора прерывания 67h */
    geninterrupt(0x21);
    EmmPtr = (char far *)MK_FP(_ES, 0x0a);
    /* Проверка имени драйвера
    на совпадение со стандартным. */
    while(* EmmName)
        if(*EmmName++ != * EmmPtr++)
            return NOTINSTALLED;
    return NOERROR;
}

```

После того, как обнаружено наличие ЕММ-драйвера, можно обращаться к различным функциям прерывания 67h. Все эти функции в регистре АН возвращают код ошибки (табл. 6.1).

Далее приводится спецификация наиболее употребительных функций EMS, определенных версиями 4.0 и старше:

АН = 40h — определение состояния EMS. Функция используется после того, как определено наличие ЕММ-драйвера. Возврат в АН нулевого значения свидетельствует о готовности EMS-платы к работе;

АН = 41h — определение сегмента, начиная с которого в памяти располагается фрейм физических страниц (возвращаемое в ВХ значение, если АН = 0);

АН = 42h — определение общего объема EMS-памяти и количества свободных для распределения страниц. На выходе из функции регистр ВХ сообщает общее количество логических страниц размером по 16К байт каждая; регистр DX сообщает общее количество еще не распределенных страниц;

Табл. 6.1. Некоторые коды ошибок из EMM-драйвера

Значение в регистре AH	Комментарий
0	Отсутствие ошибок: запрошенная функция успешно выполнена
81h	Внутренняя ошибка в EMM-драйвере
82h	EMM-драйвер занят
83h	Недопустимый EMM-префикс
84h	Запрос неописанной в стандарте функции
85h	Исчерпан лимит на EMM-префиксы
86h	Ошибка сохранения или восстановления карты отображения
87h	Запрошенное число страниц превышает объем EMS-памяти
88h	Запрошенное число страниц больше числа свободных страниц
89h	Запрос закрепления нуля страниц
8Ah	При создании карты страниц указан номер, превышающий максимально допустимый для данного EMM-префикса
8Bh	Номер физической страницы больше 3
8Ch	Область сохранения контекста карты отображения пуста
8Dh	Попытка сохранить контекст карты отображения вторично
8Eh	Попытка восстановления контекста карты без предварительного его сохранения
8Fh	Недопустимый номер подфункции в регистре AL

АН = 43h — распределение указанного в регистре ВХ числа логических страниц. В случае успеха (АН = 0) запрошенные подряд расположенные логические страницы считаются в последующих запросах распределения занятыми и не связываются с другим EMM-префиксом. Для ссылок на распределенные страницы должен использоваться EMM-префикс, возвращаемый в регистре DX;

АН = 44h — построение карты отображения. Связывает физическую страницу, номер которой задается в регистре AL (от 0 до 3), с логической страницей, номер которой задается в регистре ВХ. Логическая страница закреплена за EMM-префиксом, задаваемым в регистре DX. Нетрудно догадаться, что максимальное значение номера логической страницы не должно превышать общего числа запрошенных при распределении логических страниц. Для построения карты, отображающей весь фрейм физических страниц, требуется выполнить данную функцию отдельно для каждой из физических страниц (т.е. четыре раза подряд);

АН = 45h — закрытие EMM-префикса, заданного в регистре DX и освобождение закрепленных за ним логических страниц;

АН = 46h — определение версии EMS. Возвращаемое в регистре AL значение интерпретируется так: биты 7 - 4 задают “старший” номер, биты 3 - 0 — “младший”. Например, версии EMS 4.0 соответствует значение в AL, равное 40h.

Перечисленные функции поддерживаются всеми версиями EMS, начиная с 3.2. В версии 4.0 предусмотрен ряд усовершенствований. В частности, объем управляемой EMS-памяти увеличен до 32М байт. Размер фрейма физических страниц может равняться 128К байт, разрешая не 4, а 8 физических страниц. Допускается отображение физических страниц и на память в пределах основной памяти 1М байт, предусмотрена возможность предохранения от разрушения при “горячей” перезагрузке выбранных страниц памяти. Значительно увеличено число функций, предназначенных, прежде всего, для упрощения быстрого переключения системы с задачи на задачу, переноса больших блоков данных из основной памяти в EMS и обратно, перемещения данных в пределах EMS-памяти и др. Далее рассматриваются только некоторые из функций расширенного версией 4.0 набора управления EMS-памятью:

АН = 47h — сохранение текущей карты отображения между всеми физическими и соответствующими им логическими страницами. В регистре DX задается EMM-префикс логических страниц текущего процесса.

АН = 48h — восстановление карты отображения в состояние, в котором она находилась в момент вызова функции АН = 47h с тем же самым EMM-префиксом, задаваемым в регистре DX. Используя функции АН=47h и АН=48h с одним и тем же префиксом, TSR-программа может сохранить, а затем восстановить карту отображения фоновой программы.

Далее приводятся примеры Си-функций для управления EMS. Они образуют базовый набор операций и могут послужить отправной точкой для создания собственной более полной библиотеки. Функция EmsVersion() возвращает код версии EMS. В случае возникновения ошибки возвращается 0 и во внешнюю переменную EmsError записывается код ошибки EMM-драйвера (см. табл. 6.1).

```
/* L6_2.C
```

Определяет версию драйвера, управляющего EMS-памятью.

Возврат:

```
    unsigned int - версия EMS:
                        биты 7 - 4 - "старший" номер;
                        биты 3 - 0 - "младший" номер;
    0 - при возникновении ошибки. */
#define NOTINSTALLED 1
#include <dos.h>

int EmsInstalled(void);
int EmsVersion(void)
{ extern int EmsError;
  /* Проверка того, установлен ли ЕММ-драйвер. */
  if(EmsInstalled() == NOTINSTALLED)
  {
    EmsError = NOTINSTALLED;
    return (0);
  }
  /* Определение номера версии. */
  _AH = 0x46;
  geninterrupt(0x67);
  if(_AH)
  { EmsError = _AH;
    return (0);
  }
  EmsError = 0;
  return(_AL);
}
```

Функция EmsPagesAvail() возвращает количество страниц логической памяти, доступных для распределения. В случае возникновения ошибки возвращается -1 и во внешнюю переменную EmsError записывается код ошибки ЕММ-драйвера (см. табл. 6.1).

```
/*
L6_3.C
Определяет количество свободных
логических страниц EMS-памяти.
Возврат: 0 - число доступных для
           распределения страниц;
        -1 - возникновение ошибки. */
```



```

#define NOTINSTALLED 1
#include <dos.h>
int EmsInstalled(void);
int EmsPagesAvail(void)
{
    extern int EmsError;
    /* Проверка того, установлен ли EMM-драйвер. */
    if(EmsInstalled() == NOTINSTALLED)
    {
        EmsError = NOTINSTALLED;
        return (-1);
    }
    /* Определение числа доступных страниц. */
    _AH = 0x42;
    geninterrupt(0x67);
    if(_AH)
    {
        EmsError = _AH;
        return (-1);
    }
    EmsError = 0;
    return(_BX);
}

```

Функция EmsAlloc() распределяет Pages страниц EMS-памяти логической памяти, доступных для распределения. В случае возникновения ошибки возвращается -1 и во внешнюю переменную EmsError записывается код ошибки EMM-драйвера (см. табл. 6.1). В противном случае возвращается EMM-префикс для ссылки на выделенные страницы.

```

/*
L6_4.C
Распределяет Pages логических страниц EMS-памяти.
Возврат: 0 – EMM-префикс выделенных страниц;
        -1 – возникновение ошибки. */
#define NOTINSTALLED 1
#include <dos.h>

int EmsInstalled(void);
int EmsAlloc(Pages)

```



```

{
    extern int EmsError;
    /* Проверка того, установлен ли EMM-драйвер. */
    if(EmsInstalled() == NOTINSTALLED)
    {
        EmsError = NOTINSTALLED;
        return (-1);
    }
    /* Запрос Pages доступных страниц. */
    _BX = Pages;
    _AH = 0x43;
    geninterrupt(0x67);
    if(_AH)
    {
        EmsError = _AH;
        return (-1);
    }
    EmsError = 0; /* нет ошибки */
    return (_DX);
}

```

Точный адрес памяти, с которого в пределах первого мегабайта начинается фрейм физических страниц, возвращает функция EmsFrameAdr(). При возникновении ошибки возвращается NULL-указатель и во внешнюю переменную EmsError записывается код ошибки EMM-драйвера (см. табл. 6.1).

```

/*
    L6_5.C
    Определяет адрес фрейма физических страниц.
    Возврат: far-указатель на первый байт фрейма;
            NULL - возникновение ошибки. */

#define NOTINSTALLED 1
#include <dos.h>
int EmsInstalled(void);

char far * EmsFrameAdr(void)
{
    extern int EmsError;
    /* Проверка того, установлен ли EMM-драйвер. */

```



```

if(EmsInstalled() == NOTINSTALLED)
{
    EmsError = NOTINSTALLED;
    return (-1);
}
/* Формирование адреса фрейма физических страниц. */
_AH = 0x41;
geninterrupt(0x67);
if(_AH)
{
    EmsError = _AH;
    return((char far *) 0);
}
EmsError = 0;
return((char far *) MK_FP(_BX, 0));
}

```

Функция EmsMap() строит карту отображения для четырех логических страниц, передаваемых как параметры. В случае успеха функция возвращает 0. При возникновении ошибки возвращается -1 и во внешнюю переменную EmsError записывается код ошибки ЕММ-драйвера (см. табл. 6.1).

```

/*
L6_6.C
Задаст отображение четырех логических страниц,
номера которых помещены в массиве Pages[], и
четырёх физических страниц. Логические страницы
распределены ранее, и для ссылок на них
используется ЕММ-префикс Handle.
Возврат: 0 - в случае успеха;
        -1 -в случае ошибки. */
#define NOTINSTALLED 1
#include <dos.h>

int EmsInstalled(void);

int EmsMap(int Handle, int Pages[])
{
    extern int EmsError;
    register int i;

```



```

/* Проверка того, установлен ли EMM-драйвер. */
if(EmsInstalled() == NOTINSTALLED)
{
    EmsError = NOTINSTALLED;
    return (-1);
}
/* Цикл построения карты отображения
для четырех страниц. */
for(i = 0; i < 4; i++)
{
    _BX = Pages[i];
    _DX = Handle;
    _AH = 0x44;
    _AL = (char) i;
    geninterrupt(0x67);
    /* Анализ состояния выполненной операции. */
    if(_AH)
    {
        EmsError = _AH;
        return(-1);
    }
}
EmsError = 0;
return (0);
}

```

Функция EmsFree() освобождает все логические страницы EMS-памяти, закрепленные за EMM-префиксом Handle. В случае успеха функция возвращает 0. При возникновении ошибки возвращается -1 и во внешнюю переменную EmsError записывается код ошибки EMM-драйвера (см. табл. 6.1).

```

/*
L6_7.C
Освобождает все логические страницы EMS-памяти,
закрепленные за EMM-префиксом Handle.
Возврат: 0 – в случае успеха;
        -1 – в случае ошибки. */

```

```

#define NOTINSTALLED 1
#include <dos.h>

```



```

int EmsInstalled(void);
int EmsFree(int Handle)
{
    extern int EmsError;
    /* Проверка того, установлен ли ЕММ-драйвер. */
    if(EmsInstalled() == NOTINSTALLED)
    {
        EmsError = NOTINSTALLED;
        return (-1);
    }
    /* Освобождение логических страниц. */
    _DX = Handle;
    _AH = 0x45;
    geninterrupt(0x67);
    /* Анализ состояния выполненной операции. */
    if(_AH)
    {
        EmsError = _AH;
        return(-1);
    }
    EmsError = 0;
    return (0);
}

```

Функция EmsSave() сохраняет текущую карту отображения для заданного ЕММ-префикса Handle во внутренней области сохранения EMS-платы. В случае успеха функция возвращает 0. При возникновении ошибки возвращается -1 и во внешнюю переменную EmsError записывается код ошибки ЕММ-драйвера (см. табл. 6.1).

```

/* L6_8.C
    Сохраняет текущую карту отображения для
    заданного ЕММ-префикса Handle во внутренней
    области сохранения EMS-платы.
    Возврат:  0 – в случае успеха;
             -1 – в случае ошибки. */
#define NOTINSTALLED 1
#include <dos.h>

int EmsInstalled(void);
int EmsSave(int Handle)

```



```

{
extern int EmsError;
/* Проверка того, установлен ли EMM-драйвер. */
if(EmsInstalled() == NOTINSTALLED)
{
    EmsError = NOTINSTALLED;
    return (-1);
}
/* Сохранение карты отображения. */
_DX = Handle;
_AH = 0x47;
geninterrupt(0x67);
/* Анализ состояния выполненной операции. */
if(_AH)
{
    EmsError = _AH;
    return(-1);
}
EmsError = 0;
return (0);
}

```

Функция EmsRestore() восстанавливает из внутренней памяти EMS-адаптера сохраненную ранее карту отображения для EMM-префикса Handle.

```

/* L6_9.C
Восстанавливает карту отображения для
заданного EMM-префикса Handle из внутренней
области сохранения EMS-платы.
Возврат:  0 – в случае успеха;
          -1 – в случае ошибки. */

```

```

#define NOTINSTALLED 1
#include <dos.h>

```

```

int EmsInstalled(void);
int EmsRestore(int Handle)
{
extern int EmsError;
/* Проверка того, установлен ли EMM-драйвер. */
if(EmsInstalled() == NOTINSTALLED)

```



```

    {
        EmsError = NOTINSTALLED;
        return (-1);
    }
    /* Восстановление карты отображения. */
    _DX = Handle;
    _AH = 0x48;
    geninterrupt(0x67);
    /* Анализ состояния выполненной операции. */
    if(_AH)
    {
        EmsError = _AH;
        return(-1);
    }
    EmsError = 0;
    return (0);
}

```

Приведем небольшую демонстрационную программу для иллюстрации управления EMS-памятью и организации доступа к ней. Программа определяет наличие ЕММ-драйвера, затем собирает сведения о EMS-памяти: номер версии, общее число страниц и доступное для распределения количество логических страниц. Если последних не менее 4, демонстрационная программа продолжает свою работу. Распределяется 4 логические страницы и для них строится карта следующего вида: физическая страница 0 отображается в логическую страницу 0, физическая страница 1 — в логическую страницу 1 и т.д. В начало каждой из логических страниц записывается строка “Логическая страница номер x”, где $x = 0; 1; 2; 3$. Построенная программой карта отображения запоминается. Затем устанавливается инверсная карта: физическая страница 0 отображается в логическую страницу 3, физическая страница 1 — в логическую страницу 2 и т.д. После этого выводится содержимое строк, записанных в начале каждой физической страницы. И, наконец, демонстрационная программа восстанавливает ранее сохраненную карту отображения и повторяет операцию вывода строк.

Последним действием демонстрационная программа освобождает распределенные логические страницы.

```
/* L6_10.C
```

```
    Демонстрирует управление EMS-памятью и доступ к ней.  
    Компилируется с использованием модели памяти  
    COMPACT, LARGE, HUGE*/
```

```
#include <stdio.h>
```

```
#include <dos.h>
```

```
#define    NOTINSTALLED    1
```

```
#define    NOERROR          0
```

```
int EmsInstalled(void);
```

```
int EmsVersion(void);
```

```
int EmsPagesAvail(void);
```

```
int EmsAlloc(int Pages);
```

```
int EmsFree(int Handle);
```

```
char far * EmsFrameAdr(void);
```

```
int EmsMap(int Handle, int Pages[]);
```

```
int EmsSave(int Handle);
```

```
int EmsRestore(int Handle);
```

```
int EmsError;
```

```
int main(void)
```

```
{
```

```
    unsigned PagesOffset[4] = { 0,          /* смещения  */  
                                0x4000,     /* до начала */  
                                0x8000,     /* физических */  
                                0xC000 }; /* страниц   */
```

```
    char far * PageFrame; /* указатель на фрейм  
                           физических страниц */
```

```
    int AvailPages, Version, Handle, Pages[4], i;
```

```
    /* Проверка того, установлен ли EMM-драйвер. */
```

```
    if(EmsInstalled() == NOTINSTALLED)
```

```
    {
```

```
        puts("Отображаемая (Expanded) память отсутствует");  
        return (1);
```

```
    }
```

```
    /* Сбор сведений о EMS-памяти. */
```

```
    if(!(Version = EmsVersion()))
```



```

{
    printf("Ошибка при определении версии EMS." \
        " Код-%x\n", EmsError);
    return (2);
}
puts("В системе установлена "\
    "отображаемая (Expanded) память");
printf("Версия EMS: %d.%d\n", Version >> 4,
    Version & 0x000F);
if((AvailPages = EmsPagesAvail()) == -1)
{
    printf("Ошибка при определении числа "\
        "свободных страниц EMS." \
        " Код-%x\n", EmsError);
    puts("Ошибка при определении числа "\
        "свободных страниц EMS");
    return (3);
}
printf("Свободных логических страниц EMS: %d\n",
    AvailPages);
/* Распределение четырех логических страниц
   и определение начального адреса фрейма
   физических страниц. */
if((Handle = EmsAlloc(4)) == -1)
{
    printf("Ошибка при распределении "\
        "свободных страниц EMS" \
        " Код-%x\n", EmsError);
    return (4);
}
if((PageFrame = EmsFrameAdr()) == NULL)
{
    printf("Ошибка при определении адреса фрейма "\
        "физических страниц. Код-%x\n", EmsError);
    return (5);
}
printf("Распределены 4 логические "\
    "страницы EMS-памяти.\n" \
    "Префикс для ссылок: %d.\n" \
    "Адрес фрейма физических страниц: %p\n",
    Handle, PageFrame);

```



```

/* Построение карты отображения и ее сохранение. */
Pages[0] = 0; Pages[1] = 1;
Pages[2] = 2; Pages[3] = 3;
if(EmsMap(Handle, Pages) == -1)
{
    printf("Ошибка при построении карты отображения."\\
        " Код-%x\\n", EmsError);
    return (5);
}
if(EmsSave(Handle) == -1)
{
    printf("Ошибка при сохранении карты отображения."\\
        " Код-%x\\n", EmsError);
    return (6);
}
/* Доступ к выделенным логическим страницам:
   запись в самом начале каждой страницы тестовой
   ASCIIZ-строки. */
for(i = 0; i < 4; i++)
    sprintf(PageFrame + PagesOffset[i],
        "Логическая страница %d\\n", i);
/* Установка инверсной карты. */
Pages[0] = 3; Pages[1] = 2;
Pages[2] = 1; Pages[3] = 0;
if((EmsMap(Handle, Pages)) == -1)
{
    printf("Ошибка при построении карты отображения."\\
        " Код-%x\\n", EmsError);
    return (5);
}
/* Доступ к выделенным логическим страницам:
   вывод на экран записанной в начале страницы
   тестовой ASCIIZ-строки. */
for(i = 0; i < 4; i++)
    printf(PageFrame + PagesOffset[i]);
/* Восстановление ранее сохраненной карты. */
if(EmsRestore(Handle) == -1)
{
    printf("Ошибка при восстановлении "карты отображения."\\
        " Код-%x\\n", EmsError);
    return (7);
}

```



```

/* Доступ к выделенным логическим страницам:
   вывод записанной в начале страницы
   ASCIIZ-строки индикации. */
for(i = 0; i < 4; i++)
    printf(PageFrame + PagesOffset[i]);
/* Освобождение распределенной EMS-памяти. */
if(EmsFree(Handle) == -1)
{
    printf("Ошибка при освобождении "\
          "страниц EMS-памяти"\
          " Код-%x\n", EmsError);
    return (8);
}
return (0); /* тест успешно завершен */
}

```

Для получения .EXE-файла используется файл проекта (например, L6_10.PRJ), в который включены файлы L6_1.C — L6_10.C. При своем исполнении демонстрационная программа выдает на экран следующую информацию:

```

G:\>L6_10.EXE
В системе установлена отображаемая (Expanded) память
Версия EMS: 4.0
Свободных логических страниц EMS: 169
Распределены 4 логические страницы EMS-памяти.
Префикс для ссылок: 3.
Адрес фрейма физических страниц: E000:0000
Логическая страница 3 - используется инверсная карта
Логическая страница 2
Логическая страница 1
Логическая страница 0
Логическая страница 0 - используется исходная карта
Логическая страница 1
Логическая страница 2
Логическая страница 3

```

Рассмотренная демонстрационная программа иллюстрирует использование EMS-памяти в обычных нерезидентных программах. Работа TSR-программы с отображаемой памятью имеет некоторые особенности:

1) распределение EMS-памяти, необходимой для работы резидентной программы, должно выполняться программой инсталляции, а не при активизации резидентной части TSR;

2) сразу же после активизации резидентная часть TSR должна сохранить текущую карту отображения (например, используя приведенную ранее функцию `EmsSave()`), затем построить карту отображения для собственных логических страниц EMS-памяти, а перед выходом в фоновую программу выполнить восстановление карты (например, используя приведенную ранее функцию `EmsRestore()`). Функции `EmsSave()` и `EmsRestore()` используют один и тот же EMM-префикс, полученный инсталлирующей частью TSR при распределении страниц;

3) при “выгрузке” резидентной программы из памяти (об этом см. подробнее в 2.4) необходимо освободить все логические страницы EMS-памяти, выделенной для использования резидентной программой.

Полное описание всех функций интерфейса прикладной программы с EMS можно найти в электронных справочниках. Отметим, что EMS версии 4.0 и старше поддерживает механизм именованных EMM-префиксов, ссылка на которые осуществляется по имени префикса. Специальные функции предназначены для получения полной информации об использовании EMS: числе страниц, закрепленных за каждым EMM-префиксом, специфических атрибутах страниц и пр. Группа функций облегчает работу с картой отображения физических страниц на логические, а также упрощает своппинг между основной и EMS-памятью.

EMS-память может использоваться не только для хранения данных, но и размещения кода исполняемой программы. Ограничение состоит в том, что объем непрерывно исполняемого фрагмента программы не должен превышать 64К байт, а сам фрагмент должен завершаться передачей управления на программу-диспетчер. Диспетчер записывает код первого фрагмента программы как обычный блок данных в подряд расположенные логические страницы. Строится карта отображения, например, для первых четырех логических страниц, и управление передается на точку входа во фрагмент. После того, как

управление достигает последней инструкции фрагмента и возвращается в диспетчер, строится новая карта для следующих страниц EMS-памяти и т.д. EMS версии 4.0 включает специальные функции интерфейса, упрощающие построение программ, располагающихся при исполнении в EMS-памяти.

Отметим еще раз, что “истинная” отображаемая память требует установки в компьютер специальной платы памяти. EMS-память доступна только через фрейм физического окна. Наиболее совершенные EMS-платы могут программно изменять конфигурацию, “превращаясь” в так называемую расширенную (Extended) память, управление которой подробно рассматривается в следующем параграфе. Специальные драйверы, например XMS2EMS.SYS, поддерживают такое “превращение”. Возможна ситуация и обратного “превращения”, когда в качестве отображаемой памяти используется расширенная память и специальные драйверы программно имитируют наличие EMS-платы (например, EMM386.EXE, QEMM.SYS и др.).

Достоинствами EMS-памяти являются:

- 1) возможность расширения доступной памяти в компьютерах на базе процессора 8086;
- 2) высокая скорость доступа, обеспечиваемая аппаратной переадресацией.

6.2. Управление расширенной памятью

6.2.1. Общие положения

Расширенная память (Extended Memory) — это память сверх адресного пространства в 1М байт в компьютерах, основанных на процессорах Intel 80286, 80386, 80486. Для управления такой памятью используются аппаратные и программные средства, поддерживающие *спецификацию расширенной памяти*, или *XMS (Extended Memory Specification)*. Существуют различные ее версии. Далее рассматривается спецификация версии 2.0 и старше, разработанная совместно фирмами Microsoft, Lotus,

Intel и AST Research. На рис. 6.3 приводится распределение адресного пространства с точки зрения XMS. Использование XMS предполагает установку в системе специального драйвера, называемого далее *XMS-драйвером*, который обеспечивает интерфейс прикладной программы с расширенной памятью. Обычно это входящий в состав MS-DOS устанавливаемый драйвер HIMEM.SYS. Некоторые функции интерфейса рассматриваются в следующих параграфах.

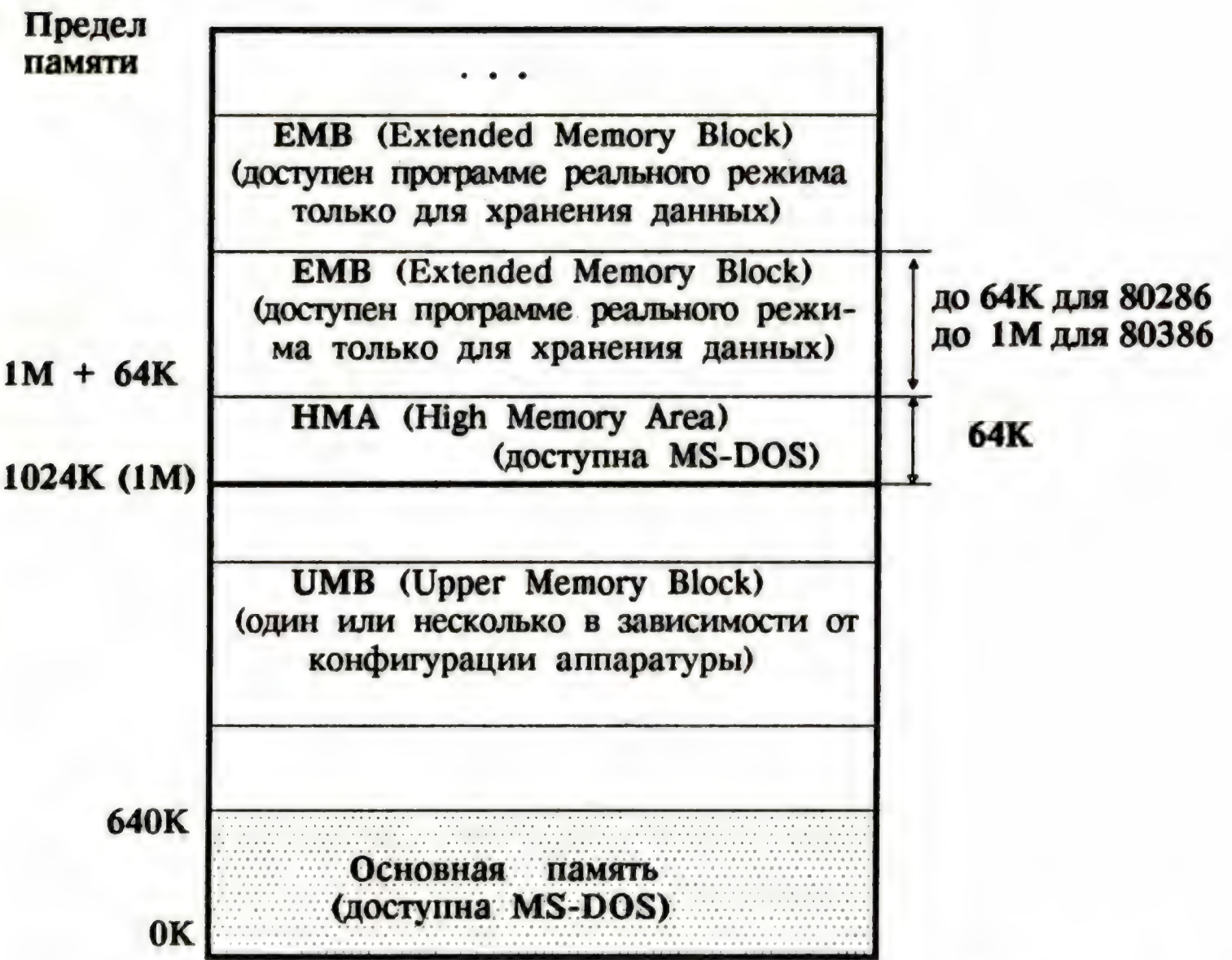


Рис.6.3. Распределение памяти в IBM PC на основе процессоров 80286/386

Кратко прокомментируем отдельные элементы, показанные на рис. 6.3. Первые 64К байт расширенной памяти (памяти, физические адреса которой превышают 1М байт) образуют HMA (High Memory Area). Доступ к HMA тре-

бует управления специальным аппаратным узлом — микросхемой 8042 в IBM PC AT. Работа микросхемы может быть заблокирована или активизирована. Выходом микросхемы является 21-й разряд физического адреса (так называемая линия A20). Если линия A20 деблокирована, НМА доступна для любой программы, работающей в реальном режиме. В этом случае перенос, возникающий при формировании 20-разрядного физического адреса, не будет игнорироваться, как это имеет место в процессоре 8086. По этой причине адресу памяти, сегментная часть которого содержит значение 0FFFFh, а смещение превышает 10h, будет соответствовать 21-разрядный физический адрес, превышающий границу в 1М байт. Однако если линия A20 заблокирована, перенос будет игнорироваться и в 80286/386. Поэтому НМА начинается с адреса FFFF:0010h и заканчивается адресом FFFF:FFFFh, а общая длина области равна 64К-16 байт.

Начиная с версии 4.0 MS-DOS, НМА может использоваться для размещения ядра операционной системы или прикладных программ. В этом случае в файл конфигурации системы помещается строка

DOS=HIGH

Обрабатывая данную команду, MS-DOS полностью захватывает всю НМА, и в дальнейшем эта область расширенной памяти управляется менеджером памяти MS-DOS. Сюда переносится резидентная часть ядра и при необходимости другие резидентные программы. При этом предполагается, что в системе уже установлен драйвер HIMEM.SYS, деблокировавший линию A20.

Блок памяти *UMB (Upper Memory Block)* доступен на некоторых компьютерах, оснащенных специальной платой так называемой *системной* или *“теневой” памяти (System или Shadow Memory)*. Общий размер этой памяти обычно составляет 256К байт. Один или несколько UMB-блоков располагаются в неиспользуемом BIOSом и видеопамью адресном пространстве выше 640К байт до границы в 1М байт. Основное назначение платы “теневой” памяти

состоит в повышении производительности за счет ускорения доступа к BIOSy. В современных компьютерах скорость физического доступа к оперативной памяти намного выше скорости доступа к ПЗУ. Поэтому в ходе начальной загрузки “основной” BIOS системы и его расширения на отдельных платах копируются в “теневую” память и в процессе работы компьютера используется копия в оперативной памяти, а не в ПЗУ. Оставшаяся свободной “теневая” память может включаться в адресное пространство MS-DOS и использоваться ею как основная и НМА-память. Таким образом, UMB расширяет общий объем памяти, доступной MS-DOS для хранения ядра и резидентных программ, оставляя в распоряжении прикладных программ практически всю основную память в 640К байт. Для создания UMB и организации интерфейса прикладной программы используются специальные устанавливаемые драйверы, например EMM386.SYS. MS-DOS версии 5.0 способна эффективно использовать UMB. Например, строка файла конфигурации

DOS = HIGH, UMB

дает указание разместить ядро операционной системы в UMB, а не в НМА.

Память *EMB* (*Extended Memory Block*) образуют один или несколько блоков, расположенных в адресном пространстве выше НМА. Для доступа к ним XMS-драйвер переключает процессор в защищенный режим. Поэтому программы реального режима могут использовать EMB только для хранения данных.

6.2.2. Функции управления расширенной памятью

Управление расширенной памятью может выполняться непосредственным обращением к XMS-драйверу либо через функции специальных библиотек. Например, компанией Microsoft распространяется Си-библиотека EMM.LIB. Она рассчитана на совместное использование с Microsoft C и в настоящий момент доступна (по крайней мере, в пре-

делах бывшего СССР) только для модели памяти SMALL. Далее рассматриваются Си-функции управления XMS-памятью, построенные на непосредственном обращении к инсталлируемому драйверу.

XMS-драйверы включают в каскад прерывания 2Fh собственную "эхо-секцию". Она, получив управление, проверяет значение регистра AH: если AH = 43h, драйвер сообщает о своем присутствии (AL=00) либо передает в регистрах ES:BX адрес точки входа в функции управления (AL=10h). Возвращаемый адрес точки входа используется в дальнейшем для непосредственного обращения к XMS-драйверу. Номер функции в этом обращении задается драйверу в регистре AH. В случае успеха XMS-драйвер возвращает AX = 0. При возникновении ошибки AX = 1, а код ошибки передается в регистре BL. Возможные коды ошибок приведены в табл. 6.2.

Отметим, что драйвер HIMEM.SYS несовместим с драйвером VDISK.SYS, который использует расширенную память по совершенно другой схеме. Поэтому, если обнаружен VDISK.SYS, многие функции управления расширенной памятью не выполняются. Например, отсутствует возможность управления HMA, блокирования или деблокирования линии A20 и др.

Первая функция, которая должна выполняться программой, использующей расширенную память, — определение присутствия в системе XMS-драйвера. Рекомендуемый стандартом способ состоит из следующих шагов:

- 1) в регистр AH записывается 43h, в AL записывается 00h и выдается прерывание 2Fh. В отличие от рассмотренного ранее метода определения наличия EMS нет нужды предварительно определять, есть ли хотя бы "заглушка" для прерывания 2Fh: MS-DOS при загрузке всегда инициализирует вектор 2Fh (о каскаде обработчиков прерывания 2Fh см. также в п.2.2.3);

- 2) анализируется возврат в регистре AL: если XMS-драйвер инсталлирован, то AL = 80h. Это, однако, не означает, что расширенная память доступна для прикладной программы;

3) определяется адрес точки входа в функции XMS-драйвера. Для этого в регистр AH записывается 43h, в AL записывается 10h и выдается прерывание 2Fh;

4) адрес точки входа в функции XMS-драйвера возвращается в регистрах ES:BX. Разумно сохранить его во внешней переменной, доступной для всех остальных Си-функций интерфейса с XMS.

Табл. 6.2. Коды ошибок управления расширенной памятью

Значение в регистре BL	Комментарий
80h	Функция не поддерживается драйвером
81h	НМА использована драйвером виртуального диска VDISK
82h	Ошибка управления линией A20
8Eh	Общая ошибка XMS-драйвера
8Fh	Невосстановимая ошибка XMS-драйвера
90h	НМА отсутствует
91h	НМА занята
92h	Объем запрошенной в НМА памяти меньше предела, установленного параметром /HMAMIN= при инсталляции XMS-драйвера HIMEM.SYS
93h	НМА не распределена
94h	Линия A20 по-прежнему деблокирована
A0h	Отсутствует свободная расширенная память
A1h	Отсутствуют свободные XMS-префиксы
A2h	Недопустимый XMS-префикс
A3h	Недопустимый XMS-префикс источника
A4h	Недопустимое смещение адреса источника
A5h	Недопустимый XMS-префикс назначения
A6h	Недопустимое смещение адреса назначения
A7h	Недопустимая длина блока
A8h	Перенос блока с недопустимым перекрытием
A9h	Ошибка паритета
AAh	Блок не “заперт”
ABh	Блок “заперт”
ACh	Переполнение счетчика числа операций “запираания” блока
ADh	Ошибка операции “запираания”
B0h	Может быть распределен UMB меньшего размера
B1h	Нет доступных UMB
B2h	Недопустимое значение сегмента адреса UMB

Здесь и далее для упрощения листингов приводимых программ используется собственный заголовочный файл “XMS.H”, в котором помещены все необходимые символические константы и прототипы функций управления расширенной памятью:


```

/***** XMS.H *****/
/*      Заголовочный файл для собственных функций      */
/*      управления расширенной памятью      */
/*****/
#if !defined (DEF_XMS)/*предотвращает повторное описание*/
/*enum XmsErrorCodes { NOERROR, NOTINSTALLED };*/
#define NOTINSTALLED 1
#define NOERROR 0
struct XmsMove
    {   unsigned long Length;
        unsigned short SourceHandle;
        unsigned long SourceOffset;
        unsigned short DestHandle;
        unsigned long DestOffset;    };
/*Прототипы функций по управлению расширенной памятью.*/
int XmsInstalled(void);
long XmsVersion(void);
int XmsRequestHMA(unsigned Space);
int XmsReleaseHMA(void);
int XmsGlobalEnableA20(void);
int XmsGlobalDisableA20(void);
int XmsLocalEnableA20(void);
int XmsLocalDisableA20(void);
int XmsQueryA20(void);
int XMSQueryLargestFree(void);
int XMSQueryTotalFree(void);
int XmsAllocateEMB(unsigned SizeK);
int XmsFreeEMB(int Handle);
int XmsMoveBlock(struct XmsMove *pMoveDescription);
long XmsLockEMB(int Handle);
int XmsUnLockEMB(int Handle);
long XmsGetHandleInfo(int Handle);
long XmsReallocateEMB(int Handle, unsigned NewSizeK);
long XmsRequestUMB(unsigned SizeP);
int XmsReleaseUMB(unsigned Segment);

#define DEF_XMS 1 /* описания используются
                  только один раз */
#endif

```

Приведем пример функции XmsInstalled(), возвращающей константу NOTINSTALLED, если XMS-драйвер не

обнаружен, либо константу NOERROR в противном случае. Если драйвер обнаружен, функция определяет адрес точки входа в функции XMS-драйвера и записывает его во внешнюю переменную XmsControl. Все последующие функции перед обращением к XMS-драйверу проверяют на равенство нулю значение XmsControl. Если XmsControl равна нулю, это означает, что XMS-драйвер не был обнаружен. Еще одна внешняя переменная, описываемая в файле L6_11.C, используется для установки кода ошибки, возникающего при исполнении функций управления расширенной памятью.

```
/* L6_11.C
```

```
    Определяет наличие драйвера, управляющего XMS-памятью,  
    и адрес точки входа в функции XMS-драйвера.
```

```
Возврат :
```

```
    NOTINSTALLED (1) – если не инсталлирован XMS-драйвер;  
    NOERROR (0)      – если обнаружен драйвер XMS-памяти.
```

```
Внешние переменные:
```

```
    XmsControl – на выходе содержит адрес входа  
                 в функции XMS-драйвера;
```

```
    XmsError   – код ошибки функций управления  
                 XMS-памятью.*/
```

```
#include <dos.h>
```

```
#include "xms.h"
```

```
void far (*XmsControl)(void); /*точка входа в XMS-драйвер*/  
unsigned XmsError; /* код ошибки */
```

```
int XmsInstalled(void)
```

```
{ /* Определение наличия XMS-драйвера. */
```

```
    _AX = 0x4300;
```

```
    geninterrupt(0x2F);
```

```
    if(_AL != 0x80)
```

```
        { XmsError = NOTINSTALLED;
```

```
          return (NOTINSTALLED); }
```

```
/*Определение адреса точки входа в функции XMS-драйвера.*/
```

```
    _AX = 0x4310;
```

```
    geninterrupt(0x2F);
```

```
    XmsControl = (void far (*)(void))MK_FP(_ES, _BX);
```

```
    XmsError = NOERROR;
```

```
    return (NOERROR);
```

```
}
```


После того, как обнаружен XMS-драйвер и определена точка входа в его функции управления, прикладная программа может управлять расширенной памятью. Номер функции задается в регистре AH. Далее даются спецификации функций управления расширенной памятью:

АН = 00h — определение номера версии XMS. Функция в регистре AX сообщает номер версии, в регистре BX — внутренний номер версии драйвера. Регистр DX указывает на наличие или отсутствие HMA: DX = 0001h, если обнаружена область HMA; DX = 0h в противном случае. Биты 15 - 8 регистра AX задают “старший” номер версии, биты 7 - 0 — “младший” номер. Например, если AX = 0200h, XMS-драйвер поддерживает стандарт 2.00. Регистр DX сообщает только о существовании, но не о доступности HMA;

АН = 01h — запрос распределения HMA. Если его выполняет инсталлятор TSR-программы или секция установки устанавливаемого драйвера, на входе в функцию в регистре DX задается размер запрашиваемого пространства HMA в байтах. Если запрос HMA выполняет обычная программа, то DX = FFFFh. Если запрос HMA удовлетворен, в регистре AX возвращается 0001h. В противном случае AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2). Существуют некоторые ограничения на использование HMA, обсуждаемые далее в п. 6.2.3;

АН = 02h — освобождение памяти в HMA. Если запрос освобождения HMA удовлетворен, в регистре AX возвращается 0001h. В противном случае AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2);

АН = 03h — глобальное деблокирование линии A20. Выполняется, если программа осуществляет доступ к HMA, предварительно распределенной для собственных нужд, и управляет HMA самостоятельно. Глобальное управление линией — это установка ее состояния “на постоянно”. После того, как доступ к HMA закончен, программа должна выполнить глобальное блокирование линии A20. Если запрос деблокирования линии A20 удовлетворен, в регистре AX возвращается 0001h. В противном случае AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2);

АН = 04h — выполняет глобальное блокирование линии A20. Это необходимо, если программа осуществляет доступ к HMA, предварительно распределенной для собственных нужд. После того, как доступ к HMA закончен, программа должна выполнить глобальное блокирование линии A20. Если запрос блокирования линии A20 удовлетворен, в регистре AX возвращается 0001h. В противном случае AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2);

АН = 05h — локальное деблокирование линии A20. Выполняется, если программа осуществляет непосредственный доступ к расширенной памяти. После того, как доступ к расширенной памяти закончен, программа должна

выполнить локальное блокирование линии A20. Если запрос деблокирования линии A20 удовлетворен, в регистре AX возвращается 0001h. В противном случае AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2);

АН = 06h — локальное блокирование линии A20. Выполняется, если программа осуществляет непосредственный доступ к расширенной памяти. После того, как доступ к расширенной памяти закончен, программа должна выполнить локальное блокирование линии A20. Если запрос блокирования линии A20 удовлетворен, в регистре AX возвращается 0001h. В противном случае AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2); *

АН = 07h — запрос состояния линии A20. Если линия физически деблокирована, функция возвращает AX = 0001h. Если линия заблокирована, то AX = 0000h. Если функция выполнена без ошибок, регистр BL равен 0. Другое значение в BL свидетельствует о возникновении ошибки и равно ее коду (см. табл. 6.2). Функция определяет состояние линии, проверяя, возникает ли “перескок” адреса через границу в 1М байт;

АН = 08h — запрос состояния расширенной памяти. В регистре AX функция возвращает размер наибольшего свободного блока расширенной памяти (EMB). Регистр DX указывает общий объем в килобайтах свободной расширенной памяти; размер HMA не включается в возвращаемое значение, даже если она полностью свободна. Если функция выполнена без ошибок, регистр BL равен 0. Другое значение в BL свидетельствует о возникновении ошибки и равно ее коду (см. табл. 6.2);

АН = 09h — распределение блока расширенной памяти (EMB), размер которого указывается в регистре DX. В случае успеха значение в регистре AX равно 0001, а в DX сообщается целое неотрицательное число — так называемый *XMS-префикс*, используемый для последующих ссылок на блок. В случае возникновения ошибки AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2). Число доступных префиксов по умолчанию равно 32, но может быть увеличено с помощью параметра /NUMHANDLES в строке DEVICE = HIMEM.SYS файла конфигурации. Если все XMS-префиксы использованы, оставшаяся свободной расширенная память недоступна системе. Поэтому блоки расширенной памяти должны быть достаточно большого размера, а неиспользуемые XMS-префиксы следует как можно скорее освобождать;

АН = 0Ah — освобождение ранее распределенного блока в расширенной памяти, на который ссылается префикс, указываемый в регистре DX. В случае успеха значение в регистре AX равно 0001. В случае возникновения ошибки AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2). Любая программа, распределявшая блоки расширенной памяти, должна перед своим завершением освободить их. В противном случае распределенная память не может использоваться системой. Блоки

расширенной памяти могут быть “заперты”, и в этом случае запрос освобождения не удовлетворяется;

АН = 0Bh — перемещает блок информации в расширенной памяти. Пара регистров DS:SI задает far-указатель на блок параметров, содержащий адреса источника и назначения (табл. 6.3). В случае успеха значение в регистре AX равно 0001. В случае возникновения ошибки AX = 0000h, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2). Чаще всего функция используется для переноса данных между основной и расширенной памятью, но может использоваться для переносов и в пределах только расширенной или только основной памяти. В случае, когда поле префикса в блоке параметров (см. табл. 6.3) равно нулю, считается, что поле смещения задает обычный far-указатель в формате Сегмент (старшее слово):Смещение (младшее слово) в пределах первого мегабайта адресного пространства. Значение длины блока должно быть четным. Для повышения скорости переноса рекомендуется задавать смещения адресов на границе слова, а для 80386 — на границе двойного слова. Если блок-источник перекрывается блоком-назначением, гарантируется корректный перенос только “вперед”, т.е. когда смещение блока-источника меньше смещения блока-назначения;

Табл. 6.3. Блок параметров для переноса информации в расширенной памяти

Смещение поля от начала блока параметров	Размер поля	Использование
0	4 байта	Размер переносимого блока. Для компьютеров на базе процессора 80286 используется младшее слово (размер блока не превышает 64К байт). Для процессора 80386 размер блока может быть до 1М байт
4	2 байта	XMS-префикс блока-источника. Если он равен 0, следующее двойное слово рассматривается как far-указатель в адресном пространстве до 1М байт
6	4 байта	Смещение от начала блока-источника.
10	2 байта	XMS-префикс блока-назначения. Если он равен 0, следующее двойное слово рассматривается как far-указатель в адресном пространстве до 1М байт
12	4 байта	Смещение от начала блока-назначения

АН = 0Ch — “запирает” блок в расширенной памяти, на который ссылается XMS-префикс, задаваемый в регистре DX. Механизм блокировок предохраняет блок от удаления из памяти в мультизадачных средах, использующих разделение блоков памяти между задачами. Для каждого из ЕМВ ведется счетчик числа операций “запирания”. Блок может быть удален из памяти только в случае, когда он “отперт” всеми задачами, его использующими. Каждая операция “запирания” увеличивает, а каждая операция “отпирания” уменьшает счетчик на единицу. В случае успеха на выходе из функции регистр AX равен 0001h и в регистрах DX:BX

возвращается 32-разрядный базовый физический адрес “запертого” блока. В случае ошибки $AX = 0000h$, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2). Значение адреса позволяет, пока блок “заперт”, осуществлять доступ к нему непосредственно, используя защищенный режим работы процессора. “Запертые” блоки должны “отпираться” как можно скорее, что дает возможность менеджерам памяти мультизадачных операционных систем использовать память рациональнее;

$АН = 0Dh$ — “отпирает” блок расширенной памяти, на который ссылается XMS-префикс, задаваемый в регистре DX . В случае успеха на выходе из функции регистр AX равен $0001h$. В случае ошибки $AX = 0000h$, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2). После того, как блок “отпирается”, 32-разрядный указатель на него, возвращенный функцией $АН = 0Ch$, становится недействительным и не должен использоваться для непосредственного доступа к блоку;

$АН = 0Eh$ — возвращает информацию о XMS-префиксе, задаваемом в регистре DX . В случае успеха регистр AX на выходе из функции равен $0001h$, BH сообщает значение счетчика операций “запирания”, BL — число свободных XMS-префиксов, DX — длину блока в килобайтах. При возникновении ошибки $AX = 0000h$, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2);

$АН = 0Fh$ — перераспределяет в расширенной памяти блок, на который ссылается XMS-префикс, заданный в регистре DX . В регистре BX указывается новый размер блока. Блок должен быть предварительно “отперт”. В случае успеха регистр AX на выходе из функции равен $0001h$. При возникновении ошибки $AX = 0000h$, а значение в регистре BL уточняет причину ошибки (см. табл. 6.2);

$АН = 10h$ — запрашивает UMB. В регистре DX задается размер запрашиваемого блока в параграфах. Если запрос удовлетворен, на выходе из функции регистр $AX = 0001h$, в регистре BX сообщается сегмент адреса выделенного блока, а в DX — его истинный размер в параграфах. UMB выравнивается на границе параграфа. В случае невозможности выполнения запроса $AX = 0000h$, а в DX указывается размер максимального доступного UMB в параграфах. Регистр BL содержит код ошибки (см. табл. 6.2);

$АН = 11h$ — освобождает выделенный ранее UMB. Регистр DX задает сегмент адреса начала UMB. Если запрос удовлетворен, на выходе из функции регистр $AX = 0001h$. В случае невозможности выполнения запроса $AX = 0000h$, а в DX указывается размер максимального доступного UMB в параграфах. Регистр BL содержит код ошибки (см. табл. 6.2).

Далее рассматриваются Си-функции управления XMS-памятью. Функция `XmsVersion()` сообщает в значении типа `long` информацию о версии XMS: младшее слово равно номеру версии, а старшее слово кодирует наличие

НМА (0, если НМА не обнаружена; 1 — в противном случае). Таким образом, возврат функцией значения, большего 0x0000FFFF, свидетельствует о наличии НМА. Функция возвращает -1, если точка входа в XMS-драйвер не инициализирована предшествующим обращением к функции XmsInstalled().

/*

L6_12.C

Определяет версию драйвера, управляющего XMS-памятью.

Возврат:

long int -

 младшее слово: версия XMS:

 биты 15 - 8 - "старший" номер;

 биты 7 - 0 - "младший" номер;

 старшее слово: 0 - если не обнаружена НМА;

 1 - в противном случае;

 -1 - если предварительно не выполнена XMSInstalled.

Внешние переменные:

 XmsControl - адрес входа в функции XMS-драйвера;

 устанавливается функцией XmsInstalled;

 XmsError - код ошибки функций управления

 XMS-памятью. */

#include "xms.h"

long XmsVersion(void)

{

 extern void far (*XmsControl)(void); /* точка входа */

 extern unsigned XmsError;

 /* Выполнена ли инициализация

 адреса входа в XMS-драйвер? */

 if((long)XmsControl == 0L)

 {

 XmsError = NOTINSTALLED;

 return (-1);

 }

 /* Определение номера версии XMS. */

 _AX= 0x0000;

 XmsControl();

 return; /* long возвращается через AX и DX */

}

Функция XmsRequestHMA() распределяет Space байтов из HMA. В случае успеха она возвращает 0. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

/*

L6_13.C

Распределяет Space байтов из HMA.

Возврат:

0 – в случае успеха;

-1 – при возникновении ошибки.

Внешние переменные:

XmsControl – адрес входа в функции XMS-драйвера;
устанавливается функцией XmsInstalled();

XmsError – код ошибки функций управления
XMS-памятью. */

#include "xms.h"

int XmsRequestHMA(unsigned Space)

{

extern void far (*XmsControl)(void); /*точка входа
в XMS-драйвер*/

extern unsigned XmsError; /* код ошибки функции */

/* Выполнена ли инициализация
адреса входа в XMS-драйвер? */

if((long)XmsControl == 0L)

{

XmsError = NOTINSTALLED;

return (-1);

}

/* Попытка распределения памяти в HMA. */

_AH = 0x01;

_DX = Space;

XmsControl();

/* Анализ кода возврата. */

if(_AX == 0)

{

XmsError = 0x00FF & _BL;

return (-1);

}

XmsError = 0;

return (0);

}

Функция `XmsReleaseHMA()` освобождает НМА. В случае успеха она возвращает 0. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную `XmsError` код ошибки (см. табл. 6.2).

```
/* L6_14.C
```

```
    Освобождает НМА.
```

```
Возврат:
```

```
    0 – в случае успеха;
```

```
    -1 – при возникновении ошибки.
```

```
Внешние переменные:
```

```
    XmsControl – адрес входа в функции XMS-драйвера;
```

```
                устанавливается функцией XmsInstalled();
```

```
    XmsError – код ошибки функций управления
```

```
                XMS-памятью. */
```

```
#include "xms.h"
```

```
int XmsReleaseHMA(void)
```

```
{
```

```
    extern void far (*XmsControl)(void); /*точка входа  
                                           в XMS-драйвер*/
```

```
    extern unsigned XmsError; /* код ошибки функции */
```

```
    /* Выполнена ли инициализация адреса  
       входа в XMS-драйвер? */
```

```
    if((long)XmsControl == 0L)
```

```
    {
```

```
        XmsError = NOTINSTALLED;
```

```
        return (-1);
```

```
    }
```

```
    /* Попытка освобождения НМА. */
```

```
    _AH = 0x02;
```

```
    XmsControl();
```

```
    /* Анализ кода возврата. */
```

```
    if(_AX == 0)
```

```
    {
```

```
        XmsError = 0x00FF & _BL;
```

```
        return (-1);
```

```
    }
```

```
    XmsError = 0;
```

```
    return (0);
```

```
}
```


Функция XmsGlobalEnableA20() выполняет глобальное деблокирование линии A20. В случае успеха она возвращает 0. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

/* L6_15.C

Выполняет глобальное деблокирование линии A20.

Возврат:

0 – в случае успеха;

-1 – при возникновении ошибки.

Внешние переменные:

XmsControl – адрес входа в функции XMS-драйвера;
устанавливается функцией XmsInstalled();

XmsError – код ошибки функций управления
XMS-памятью. */

#include "xms.h"

int XmsGlobalEnableA20(void)

```
{
    extern void far (*XmsControl)(void); /*точка входа
                                           в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    /* Выполнена ли инициализация адреса
       входа в XMS-драйвер? */
    if((long)XmsControl == 0L)
    {
        XmsError = NOTINSTALLED;
        return (-1);
    }
    /* Попытка деблокирования линии A20. */
    _AH = 0x03;
    XmsControl();
    /* Анализ кода возврата. */
    if(_AX == 0)
    {
        XmsError = 0x00FF & _BL;
        return (-1);
    }
    XmsError = 0;
    return (0);
}
```


Функция XmsGlobalDisableA20() выполняет глобальное блокирование линии A20. В случае успеха она возвращает 0. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

```
/* L6_16.C
```

```
    Выполняет глобальное блокирование линии A20.
```

```
Возврат:
```

```
    0 – в случае успеха;
```

```
    -1 – при возникновении ошибки.
```

```
Внешние переменные:
```

```
    XmsControl – адрес входа в функции XMS-драйвера;
```

```
                устанавливается функцией XmsInstalled();
```

```
    XmsError   – код ошибки функций управления
```

```
                XMS-памятью. */
```

```
#include "xms.h"
```

```
int XmsGlobalDisableA20(void)
```

```
{
```

```
    extern void far (*XmsControl)(void); /*точка входа  
                                           в XMS-драйвер*/
```

```
    extern unsigned XmsError; /* код ошибки функции */
```

```
    /* Выполнена ли инициализация адреса  
       входа в XMS-драйвер? */
```

```
    if((long)XmsControl == 0L)
```

```
    {
```

```
        XmsError = NOTINSTALLED;
```

```
        return (-1);
```

```
    }
```

```
    /* Попытка блокирования линии A20. */
```

```
    _AH = 0x04;
```

```
    XmsControl();
```

```
    /* Анализ кода возврата. */
```

```
    if(_AX == 0)
```

```
    {
```

```
        XmsError = 0x00FF & _BL;
```

```
        return (-1);
```

```
    }
```

```
    XmsError = 0;
```

```
    return (0);
```

```
}
```


Функции XmsLocalEnableA20() и XmsLocalDisableA20() выполняют соответственно локальное деблокирование и блокирование линии A20. В случае успеха функции возвращают 0. При возникновении ошибки они возвращают -1 и записывают во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

/*

L6_17.C

Выполняет локальное деблокирование линии A20.

Возврат:

0 – в случае успеха;

-1 – при возникновении ошибки.

Внешние переменные:

XmsControl – адрес входа в функции XMS-драйвера;
устанавливается функцией XmsInstalled();

XmsError – код ошибки функций управления
XMS-памятью. */

#include "xms.h"

int XmsLocalEnableA20(void)

{

extern void far (*XmsControl)(void); /*точка входа
в XMS-драйвер*/

extern unsigned XmsError; /* код ошибки функции */

/* Выполнена ли инициализация адреса
входа в XMS-драйвер? */

if((long)XmsControl == 0L)

{

XmsError = NOTINSTALLED;

return (-1);

}

/* Попытка деблокирования линии A20. */

_AH = 0x05;

XmsControl();

/* Анализ кода возврата. */

if(_AX == 0)

{

XmsError = 0x00FF & _BL;

return (-1);

}

XmsError = 0;


```

    return (0);
}

/*
    L6_18.C
    Выполняет локальное блокирование линии A20.
Возврат:
    0 – в случае успеха;
    -1 – при возникновении ошибки.
Внешние переменные:
    XmsControl – адрес входа в функции XMS-драйвера;
                  устанавливается функцией XmsInstalled();
    XmsError   – код ошибки функций управления
                  XMS-памятью. */

#include "xms.h"
int XmsLocalDisableA20(void)
{
    extern void far (*XmsControl)(void); /*точка входа
                                          в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    /* Выполнена ли инициализация адреса
       входа в XMS-драйвер? */
    if((long)XmsControl == 0L)
    {
        XmsError = NOTINSTALLED;
        return (-1);
    }
    /* Попытка блокирования линии A20. */
    _AH = 0x06;
    XmsControl();
    /* Анализ кода возврата. */
    if(_AX == 0)
    {
        XmsError = 0x00FF & _BL;
        return (-1);
    }
    XmsError = 0;
    return (0);
}

```


Функция XmsQueryA20() определяет состояние линии A20 и возвращает 1, если линия деблокирована, 0 — если линия A20 блокирована. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

```
/* L6_19.C
```

```
    Определяет текущее состояние линии A20.
```

```
Возврат:
```

```
    1 — линия деблокирована;
```

```
    0 — линия блокирована;
```

```
    -1 — при возникновении ошибки.
```

```
Внешние переменные:
```

```
    XmsControl — адрес входа в функции XMS-драйвера;
```

```
                устанавливается функцией XmsInstalled();
```

```
    XmsError — код ошибки функций управления
```

```
                XMS-памятью. */
```

```
#include "xms.h"
```

```
int XmsQueryA20(void)
```

```
{
```

```
    extern void far (*XmsControl)(void); /*точка входа  
                                           в XMS-драйвер*/
```

```
    extern unsigned XmsError; /* код ошибки функции */
```

```
    /* Выполнена ли инициализация адреса
```

```
        входа в XMS-драйвер? */
```

```
    if((long)XmsControl == 0L)
```

```
    {
```

```
        XmsError = NOTINSTALLED;
```

```
        return (-1);
```

```
    }
```

```
    /* Попытка выполнения функции. */
```

```
    _AH = 0x07;
```

```
    XmsControl();
```

```
    /* Анализ кода возврата. */
```

```
    if(_BL != 0)
```

```
    {
```

```
        XmsError = 0x00FF & _BL;
```

```
        return (-1);
```

```
    }
```

```
    XmsError = 0;
```

```
    return (_AX);
```

```
}
```


Приводимые далее функции XMSQueryLargestFree() и XMSQueryTotalFree() возвращают соответственно размер наибольшего свободного блока и общий объем свободной расширенной памяти. При возникновении ошибки функции возвращают -1 и записывают во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

```
/*
  L6_20.C
  Определяет размер наибольшего свободного
  блока расширенной памяти.
Возврат :
  >= 0 – размер наибольшего блока в килобайтах;
  -1 – при возникновении ошибки.
Внешние переменные:
  XmsControl – адрес входа в функции XMS-драйвера;
                устанавливается функцией XmsInstalled();
  XmsError   – код ошибки функций управления
                XMS-памятью. */
```

```
#include "xms.h"
int XMSQueryLargestFree(void)
{
    extern void far (*XmsControl)(void); /*точка входа
                                           в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    /* Выполнена ли инициализация адреса входа
       в XMS-драйвер? */
    if((long)XmsControl == 0L)
    {
        XmsError = NOTINSTALLED;
        return (-1);
    }
    /* Попытка выполнения функции. */
    _AH = 0x08;
    XmsControl();
    /* Анализ кода возврата. */
    if(_BL != 0)
    {
        XmsError = 0x00FF & _BL;
        return (-1);
    }
}
```



```

XmsError = 0;
return (_AX);
}

/*
    L6_21.C
    Определяет общий объем свободной расширенной памяти.
    Возврат:
    >= 0 – объем свободной памяти в килобайтах;
    -1 – при возникновении ошибки.
    Внешние переменные:
    XmsControl – адрес входа в функции XMS-драйвера;
                  устанавливается функцией XmsInstalled();
    XmsError   – код ошибки функций управления
                  XMS-памятью. */

```

```

#include "xms.h"
int XMSQueryTotalFree(void)
{
    extern void far (*XmsControl)(void); /*точка входа
                                           в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    /* Выполнена ли инициализация адреса
       входа в XMS-драйвер? */
    if((long)XmsControl == 0L)
    {
        XmsError = NOTINSTALLED;
        return (-1);
    }
    /* Попытка выполнения функции. */
    _AH = 0x08;
    XmsControl();
    /* Анализ кода возврата. */
    if(_BL != 0)
    {
        XmsError = 0x00FF & _BL;
        return (-1);
    }
    XmsError = 0;
    return (_DX);
}

```


Функция XmsAllocateEMB() распределяет в расширенной памяти блок размером SizeK килобайтов. В случае успеха возвращает XMS-префикс. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

```
/* L6_22.C
```

```
    Распределяет в расширенной памяти блок  
    размером SizeK килобайтов.
```

```
Возврат:
```

```
    >0 – XMS-префикс для ссылок на выделенный блок;  
    -1 – при возникновении ошибки.
```

```
Внешние переменные:
```

```
    XmsControl – адрес входа в функции XMS-драйвера;  
                  устанавливается функцией XmsInstalled();
```

```
    XmsError   – код ошибки функций управления  
                  XMS-памятью. */
```

```
#include "xms.h"
```

```
int XmsAllocateEMB(unsigned SizeK)
```

```
{
```

```
    extern void far (*XmsControl)(void); /*точка входа  
                                           в XMS-драйвер*/
```

```
    extern unsigned XmsError; /* код ошибки функции */
```

```
    /* Выполнена ли инициализация адреса
```

```
        входа в XMS-драйвер? */
```

```
    if((long)XmsControl == 0L)
```

```
    {
```

```
        XmsError = NOTINSTALLED;
```

```
        return (-1);
```

```
    }
```

```
    /* Попытка выполнения функции. */
```

```
    _AH = 0x09;
```

```
    _DX = SizeK;
```

```
    XmsControl();
```

```
    /* Анализ кода возврата. */
```

```
    if(_AX == 0)
```

```
    { XmsError = 0x00FF & _BL;
```

```
        return (-1);
```

```
    }
```

```
    XmsError = 0;
```

```
    return (_DX);
```

```
}
```


Функция XmsFreeEMB(Handle) освобождает блок расширенной памяти, на который ссылается XMS-префикс Handle. В случае успеха она возвращает 0. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

```
/* L6_23.C
```

```
    Освобождает в расширенной памяти блок,  
    на который ссылается Handle.
```

```
Возврат:
```

```
    0   – блок освобожден;  
    -1  – при возникновении ошибки.
```

```
Внешние переменные:
```

```
    XmsControl – адрес входа в функции XMS-драйвера;  
                  устанавливается функцией XmsInstalled();  
    XmsError   – код ошибки функций управления  
                  XMS-памятью. */
```

```
#include "xms.h"
```

```
int XmsFreeEMB(int Handle)
```

```
{
```

```
    extern void far (*XmsControl)(void); /*точка входа  
                                           в XMS-драйвер*/
```

```
    extern unsigned XmsError; /* код ошибки функции */
```

```
    /* Выполнена ли инициализация адреса  
       входа в XMS-драйвер? */
```

```
    if((long)XmsControl == 0L)
```

```
    {
```

```
        XmsError = NOTINSTALLED;
```

```
        return (-1);
```

```
    }
```

```
    /* Попытка выполнения функции. */
```

```
    _AH = 0x0A;
```

```
    _DX = Handle;
```

```
    XmsControl();
```

```
    /* Анализ кода возврата. */
```

```
    if(_AX == 0)
```

```
    { XmsError = 0x00FF & _BL;
```

```
        return (-1);
```

```
    }
```

```
    XmsError = 0;
```

```
    return (0);
```

```
}
```


Функция `XmsMoveBlock()` перемещает блок с одного места памяти в другое. Адреса источника и назначения, а также размер блока задаются структурной переменной по шаблону `XmsMove`, на которую указывает `pMoveDescription`. Шаблон структуры `XmsMove` приведен в заголовочном файле `XMS.H` и соответствует структуре блока параметров табл. 6.3. При возникновении ошибки функция возвращает `-1` и записывает во внешнюю переменную `XmsError` код ошибки (см. табл. 6.2).

```
/*
L6_24.C
Перемещает блок; адрес источника, назначения и
размер задает структура, на которую указывает
pMoveDescription. Неприменима в HUGE-модели.
Возврат:
0 - блок перенесен;
-1 - при возникновении ошибки.
Внешние переменные:
XmsControl - адрес входа в функции XMS-драйвера;
              устанавливается функцией XmsInstalled();
XmsError    - код ошибки функций управления
              XMS-памятью. */
```

```
#include "xms.h"
#include <dos.h>
int XmsMoveBlock(struct XmsMove *pMoveDescription)
{
    extern void far (*XmsControl)(void); /*точка входа
                                           в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    /* Выполнена ли инициализация адреса
       входа в XMS-драйвер? */
    if((long)XmsControl == 0L)
    {
        XmsError = NOTINSTALLED;
        return (-1);
    }
    /* Попытка выполнения функции. */
    _AH = 0x0B;
    _SI = FP_OFF(pMoveDescription);
    _AH = 0x0B;
```



```

XmsControl();
/* Анализ кода возврата. */
if(_AX == 0)
{
    XmsError = 0x00FF & _BL;
    return (-1);
}
XmsError = 0;
return (0);
}

```

Функция XmsLockEMB() “запирает” в расширенной памяти блок, на который ссылается XMS-префикс Handle. В случае успеха она возвращает базовый физический адрес начала блока (биты 0 - 30). Знаковый (31-й) бит возвращаемого значения является индикатором правильности работы функции. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

```

/*
L6_25.C
“Запирает” в расширенной памяти блок,
на который ссылается XMS-префикс Handle.
Возврат:
long    – базовый физический адрес начала блока;
-1      – при возникновении ошибки.
Внешние переменные:
XmsControl – адрес входа в функции XMS-драйвера;
            устанавливается функцией XmsInstalled();
XmsError   – код ошибки функций управления
            XMS-памятью. */

```

```

#include "xms.h"
long XmsLockEMB(int Handle)
{
    extern void far (*XmsControl)(void); /*точка входа
                                           в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    unsigned dx, bx;
    /* Выполнена ли инициализация адреса
       входа в XMS-драйвер? */

```



```

if((long)XmsControl == 0L)
{
    XmsError = NOTINSTALLED;
    return (-1);
}
/* Попытка выполнения функции. */
_AH = 0x0C;
_DX = Handle;
XmsControl();
/* Анализ кода возврата. */
dx = _DX; bx = _BX;
if(_AX == 0)
{
    XmsError = 0x00FF & _BL;
    return (-1);
}
XmsError = 0;
return (((long) (dx & 0x7F) << 16) | (unsigned) bx);
}

```

Функция XmsUnLockEMB() “отпирает” в расширенной памяти блок, на который ссылается XMS-префикс Handle. В случае успеха она возвращает 0. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

```

/*
    L6_26.C
    “Отпирает” в расширенной памяти блок,
    на который ссылается XMS-префикс Handle.
Возврат:
    0 – блок “отперт”;
    -1 – при возникновении ошибки.
Внешние переменные:
    XmsControl – адрес входа в функции XMS-драйвера;
                  устанавливается функцией XmsInstalled();
    XmsError   – код ошибки функций управления
                  XMS-памятью. */

#include “xms.h”
int XmsUnLockEMB(int Handle)

```



```

{
extern void far (*XmsControl)(void); /*точка входа
                                     в XMS-драйвер*/
extern unsigned XmsError; /* код ошибки функции */
/* Выполнена ли инициализация адреса
   входа в XMS-драйвер? */
if((long)XmsControl == 0L)
{
    XmsError = NOTINSTALLED;
    return (-1);
}
/* Попытка выполнения функции. */
_AH = 0x0D;
_DX = Handle;
XmsControl();
/* Анализ кода возврата. */
if(_AX == 0)
{
    XmsError = 0x00FF & _BL;
    return (-1);
}
XmsError = 0;
return (0);
}

```

Функция XmsGetHandleInfo() возвращает в long информацию о блоке расширенной памяти, на который ссылается XMS-префикс Handle. Биты 30 - 16 задают размер блока в килобайтах, биты 15 - 8 — значение счетчика операций “запираения”, биты 7 - 0 содержат число свободных XMS-префиксов в системе. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

/* L6_27.C

Возвращает информацию о блоке в расширенной памяти, на который ссылается XMS-префикс Handle.

Возврат:

long: биты 30 - 16 - длина блока;
 15 - 8 - счетчик операций “запираения”;
 7 - 0 - число свободных префиксов;
 -1 - при возникновении ошибки.

Внешние переменные:

XmsControl – адрес входа в функции XMS-драйвера;
устанавливается функцией XmsInstalled();
XmsError – код ошибки функций управления
XMS-памятью. */

```
#include "xms.h"
long XmsGetHandleInfo(int Handle)
{
    extern void far (*XmsControl)(void); /*точка входа
                                           в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    unsigned bx, dx;
    /* Выполнена ли инициализация адреса
       входа в XMS-драйвер? */
    if((long)XmsControl == 0L)
    {
        XmsError = NOTINSTALLED;
        return (-1);
    }
    /* Попытка выполнения функции. */
    _AH = 0x0E;
    _DX = Handle;
    XmsControl();
    bx = _BX; dx = _DX;
    /* Анализ кода возврата. */
    if(_AX == 0)
    {
        XmsError = 0x00FF & _BL;
        return (-1);
    }
    XmsError = 0;
    return (((long) dx << 16) | (unsigned) bx);
}
```

Функция XmsReallocateEMB() повторно распределяет EMB, на который ссылается XMS-префикс Handle, устанавливая новый размер блока NewSizeK. В случае успеха она возвращает 0. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

/*

L6_28.C

Перераспределяет в расширенной памяти блок,
на который ссылается XMS-префикс Handle,
устанавливая новый размер блока NewSizeK.

Возврат:

0 – блоку задан новый размер;

-1 – при возникновении ошибки.

Внешние переменные:

XmsControl – адрес входа в функции XMS-драйвера;
устанавливается функцией XmsInstalled();

XmsError – код ошибки функций управления
XMS-памятью. */

#include "xms.h"

long XmsReallocateEMB(int Handle, unsigned NewSizeK)

{

extern void far (*XmsControl)(void); /*точка входа
в XMS-драйвер*/

extern unsigned XmsError; /* код ошибки функции */

/* Выполнена ли инициализация адреса
входа в XMS-драйвер? */

if((long)XmsControl == 0L)

{

XmsError = NOTINSTALLED;

return (-1);

}

/* Попытка выполнения функции. */

_AH = 0x0F;

_DX = Handle;

_BX = NewSizeK;

XmsControl();

/* Анализ кода возврата. */

if(_AX == 0)

{

XmsError = 0x00FF & _BL;

return (-1);

}

XmsError = 0;

return (0);

}

Функция XmsRequestUMB() пытается распределить UMB размером SizeP параграфов. В случае успеха возвращает положительное число (long), младшее слово которого содержит сегмент начала выделенного UMB, а старшее слово — фактический размер UMB. При невозможности удовлетворения запроса функция возвращает отрицательное число, младшее слово которого указывает размер наибольшего доступного UMB. Код ошибки записывается во внешнюю переменную XmsError.

```
/*
    L6_29.C
    Запрашивает UMB размером SizeP.
    Возврат:
        long >0: биты 31-16 – фактический размер
                    выделенного UMB;
                    биты 15- 0 – сегмент начала
                    выделенного UMB;
        <0: биты 15- 0 – размер наибольшего
                    доступного UMB.
    Внешние переменные:
        XmsControl – адрес входа в функции XMS-драйвера;
                    устанавливается функцией XmsInstalled();
        XmsError   – код ошибки функций управления
                    XMS-памятью. */

#include "xms.h"
long XmsRequestUMB(unsigned SizeP)
{
    extern void far (*XmsControl)(void); /*точка входа
                                           в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    /* Выполнена ли инициализация адреса
       входа в XMS-драйвер? */
    if((long)XmsControl == 0L)
    {
        XmsError = NOTINSTALLED;
        return (-1);
    }
    /* Попытка выполнения функции. */
    _AH = 0x10;
    _DX = SizeP;
```



```

XmsControl();
/* Анализ кода возврата. */
if(_AX == 0)
{
    XmsError = 0x00FF & _BL;
    return (0x0000FFFF | (long)_DX);
}
XmsError = 0;
return (((long)_DX) << 16) | _BX);
}

```

Функция XmsReleaseUMB() освобождает UMB, сегмент начала которого задает параметр Segment. В случае успеха функция возвращает 0. При возникновении ошибки функция возвращает -1 и записывает во внешнюю переменную XmsError код ошибки (см. табл. 6.2).

```

/*
L6_30.C
Освобождает UMB, начинающийся с параграфа Segment.
Возврат:
0 – UMB освобожден;
-1 – при возникновении ошибки.
Внешние переменные:
XmsControl – адрес входа в функции XMS-драйвера;
               устанавливается функцией XmsInstalled();
XmsError    – код ошибки функций управления
               XMS-памятью. */

```

```

#include "xms.h"
int XmsReleaseUMB(unsigned Segment)
{
    extern void far (*XmsControl)(void); /*точка входа
                                           в XMS-драйвер*/
    extern unsigned XmsError; /* код ошибки функции */
    /* Выполнена ли инициализация адреса
       входа в XMS-драйвер? */
    if((long)XmsControl == 0L)
    {
        XmsError = NOTINSTALLED;
        return (-1);
    }
}

```



```

/* Попытка выполнения функции. */
_AH = 0x11;
_DX = Segment;
XmsControl();
/* Анализ кода возврата. */
if(_AX == 0)
{
    XmsError = 0x00FF & _BL;
    return (-1);
}
XmsError = 0;
return (0);
}

```

Приведем демонстрационную программу для тестирования разработанных функций управления расширенной памятью и иллюстрации основных приемов работы с ней. Прежде всего программа определяет наличие XMS-драйвера и версию XMS, им реализуемую. Затем выполняется распределение блока памяти размером 100К байт и определяются параметры выделенного блока. Используя EMB, программа переносит 80 байт из основной памяти в расширенную, а затем из расширенной в основную. Перенесенная из расширенной памяти информация (DestStr) выводится на экран. Программа завершается освобождением выделенного блока. Если этого не сделать, блок остается занятым и после завершения программы.

```

/*
    L6_31.C
    Демонстрирует управление XMS-памятью и доступ к ней. */

#include <stdio.h>
#include <dos.h>
#include "xms.h"

int main(void)
{
    extern unsigned XmsError;
    int XmsHandle, TotalFreeK, LargestFreeK, UmbSize;
    long Ret, BaseAdr;

```



```

unsigned Version;
char SourceStr[80] =
    "Тестовая строка для переноса через EMB";
char DestStr[80];
struct XmsMove Parameters;
/* Проверка того, установлен ли XMS-драйвер. */
if(XmsInstalled() == NOTINSTALLED)
{
    puts("Отсутствует драйвер "\
        "для управления расширенной памятью");
    return (1);
}
/* Сбор сведений о XMS-памяти. */
Ret = XmsVersion(); Version = (int) Ret;
puts("В системе установлена "\
    "расширенная (Extended) память");
printf("Версия XMS: %d.%d\n",
    Version >> 8, Version & 0x00FF);

if(Ret > 0x0000FFFF1)
    puts("Обнаружена HMA (High Memory Area)");
else
    puts("Не обнаружена HMA (High Memory Area)");

if((TotalFreeK = XMSQueryTotalFree()) == -1)
    printf("Ошибка при определении общего объема "\
        "свободной расширенной памяти. Код-%x\n",
        XmsError);
else
    printf("Общий объем свободной "\
        "расширенной памяти: %dK.\n",
        TotalFreeK);

if((LargestFreeK = XMSQueryLargestFree()) == -1)
    printf("Ошибка при определении "\
        "максимального свободного "\
        "блока расширенной памяти. Код-%x\n",
        XmsError);
else
    printf("Максимальный свободный блок "\
        "расширенной памяти: %dK.\n",
        LargestFreeK);

```



```

/*
    Распределение блока 100K байт в расширенной памяти,
    определение его базового адреса, перенос через него
    тестовой строки SourceStr в массив DestStr[].
*/
if((XmsHandle = XmsAllocateEMB(100)) == -1)
    printf("Ошибка при распределении EMB. "\
        "Код-%x\n", XmsError);
else
{
    /* Получение информации о
        блоке в расширенной памяти. */
    if((BaseAdr = XmsLockEMB(XmsHandle)) == -1)
        printf("Ошибка при \\"запирании\\" EMB. Код-%x\n",
            XmsError);
    else
        printf("Базовый адрес блока EMB: %#010lx\n",
            BaseAdr);
    if((Ret = XmsGetHandleInfo(XmsHandle)) == -1)
        printf("Ошибка при получении информации о EMB."
            "Код-%x\n", XmsError);
    else
        printf("Длина блока: %dK. "\
            "Счетчик операций \\"запирания\":"%d\n"\
            "Число свободных XMS-префиксов:%d\n",
            (int)((Ret & 0xFFFF0000) >> 16),
            (int)(Ret & 0x0000FF00) >> 8,
            (int)Ret & 0x000000FF);
}
/*
    Перенос блока из основной
    в расширенную память.
*/
Parameters.Length = 80;
Parameters.SourceHandle = 0;
Parameters.SourceOffset =
    (unsigned long)(SourceStr);
Parameters.DestHandle = XmsHandle;
Parameters.DestOffset = 0L;
if(XmsMoveBlock(&Parameters))
    printf("Ошибка при переносе блока памяти. "\
        "Код-%x\n", XmsError);

```



```

/*
    Перенос блока из расширенной
    в основную память.
*/
Parameters.Length = 80;
Parameters.SourceHandle = XmsHandle;
Parameters.SourceOffset = 0L;
Parameters.DestHandle = 0L;
Parameters.DestOffset = (unsigned long)(DestStr);
if(XmsMoveBlock(&Parameters))
printf("Ошибка при переносе блока памяти. "\
      "Код-%x\n", XmsError);
/* Вывод строки, перенесенной через EMS. */
puts(DestStr);
/*
    Повторное распределение блока
    с новым размером 200K байт.
*/
if(XmsUnLockEMB(XmsHandle) == -1)
    printf("Ошибка при \"отпирании\" EMB. "\
          "Код-%x\n", XmsError);
else
{
    puts("Перераспределение блока "\
        "расширенной памяти.");
    if(XmsReallocateEMB(XmsHandle, 200) == -1)
        printf("Ошибка при перераспределении EMB." "\
              "Код-%x\n", XmsError);
    if((BaseAdr = XmsLockEMB(XmsHandle)) == -1)
        printf("Ошибка при \"запирании\" EMB. "\
              "Код-%x\n", XmsError);
    else
        printf("Новый базовый адрес блока EMB: "\
              "%#010lx\n", BaseAdr);
}
/*
    Освобождение выделенного блока памяти.
    Требуется обязательного \"отпирания\" блока.
*/
if(XmsUnLockEMB(XmsHandle) == -1)
    printf("Ошибка при \"отпирании\" EMB. Код-%x\n",
          XmsError);

```



```

        if(XmsFreeEMB(XmsHandle))
            printf("Ошибка при освобождении блока памяти."\"
                "Код-%x\\n",    XmsError);
        else
            puts("Распределенный блок EMB освобожден");
    }
    return (0); /* тест завершен */
}

```

Для получения .EXE-файла используется файл проекта (например, L6_31.PRJ), в который включены файлы L6_11.C, L6_12.C, L6_20.C - L6_28.C и L6_31.C. Далее приведен пример исполнения программы при одном из вариантов ее запуска на исполнение:

```
G:\>L6_31.EXE
```

В системе установлена расширенная (Extended) память

Версия XMS: 2.0

Обнаружена HMA (High Memory Area)

Общий объем свободной расширенной памяти: 1563К.

Максимальный свободный блок расширенной памяти: 1563К.

Базовый адрес блока EMB: 0x00279400

Длина блока: 100К. Счетчик операций "запирания":1

Число свободных XMS-префиксов:28

Тестовая строка для переноса через EMB - строка

DestStr[]

Перераспределение блока расширенной памяти.

Новый базовый адрес блока EMB: 0x00279400

Распределенный блок EMB освобожден

6.2.3. Особенности использования HMA

Блок памяти HMA может использоваться для хранения ядра MS-DOS версии 4.0 и старше. Для этого необходимо установить в системе через файл конфигурации CONFIG.SYS драйвер HIMEM.SYS и информировать MS-DOS о необходимости использования HMA. Делается это включением в файл конфигурации двух строк:


```
DEVICE = C:\HIMEM.SYS
```

```
DOS = HIGH
```

Операционная система при загрузке занимает всю НМА. Если ядро MS-DOS размещается в основной памяти или UMB-блоке (см. далее), НМА может использоваться для хранения данных или резидентных программ. При этом следует учитывать некоторые особенности НМА:

1) в MS-DOS не могут быть переданы far-указатели на данные, хранимые в НМА, так как MS-DOS нормализует переданные ей указатели. В частности, невозможен перенос данных на диск непосредственно из НМА; по этой же причине на НМА не должны указывать векторы прерывания установленных резидентных программ;

2) драйверы и резидентные программы должны использовать большую часть НМА; использованная однажды НМА остается недоступной для других программ до тех пор, пока не будет освобождена;

3) использование НМА требует, чтобы линия A20 была деблокирована. Поэтому обработчики прерываний не должны размещаться в НМА, так как возможна ситуация блокирования линии фоновой программой. Наиболее типичное построение TSR, использующих НМА таково: обработчики прерываний размещаются в основной памяти; в НМА помещаются данные резидентной программы и резидентные функции. Получив управление, обработчики запоминают состояние линии A20, деблокируют ее и передают управление на код, хранимый в НМА. Перед завершением выполняется восстановление состояния линии A20. Загрузку в НМА должна выполнять инсталлирующая часть TSR.

Далее приводится пример тестовой программы, распределяющей для собственных нужд НМА, записывающей туда данные и выводящей их затем на экран:

```
/*
```

```
  L6_32.C
```

```
  Демонстрирует использование НМА. */
```

```
#include <stdio.h>
```

```
#include <dos.h>
```



```

#include <mem.h>
#include "xms.h"

int main(void)
{
    extern unsigned XmsError;
    int oldA20;
    long Ret;
    unsigned Version;
    char SourceStr[80] =
        "Тестовая строка для переноса через HMA";
    char DestStr[80];
    /* Проверка того, установлен ли XMS-драйвер. */
    if(XmsInstalled() == NOTINSTALLED)
    {
        puts("Отсутствует драйвер "\
            "для управления расширенной памятью");
        return (1);
    }
    /* Проверка наличия HMA. */
    Ret = XmsVersion(); Version = (int) Ret;
    puts("В системе установлена расширенная "\
        " (Extended) память");
    printf("Версия XMS: %d.%d\n", Version >> 8,
        Version & 0x00FF);
    if(Ret > 0x0000FFFF1)
        puts("Обнаружена HMA (High Memory Area)");
    else
    {
        puts("Не обнаружена HMA (High Memory Area). "\
            "Завершение.");
        return (2);
    }
    /* Запрос состояния линии A20 и ее деблокирование
       в случае необходимости. */
    if(!(oldA20 = XmsQueryA20()))
    {
        puts("Линия A20 блокирована. "\
            "Попытка деблокирования ");
        if(XmsGlobalEnableA20())
        {
            puts("завершилась неудачей. Завершение теста.");
        }
    }
}

```



```

        return(3);
    }
}
/* Распределение 20К байт из НМА и перенос через нее
   тестовой строки SourceStr в массив DestStr[]. */
if(XmsRequestHMA(20*1024))
{
    printf("Ошибка при распределении НМА. Код-%x\n",
           XmsError);
    if(!oldA20)
        XmsGlobalDisableA20(); /* восстановление
                                состояния A20 */
    return(4);
}
/* Перенос блока информации
   из основной памяти в НМА. */
movedata(FP_SEG(SourceStr),
          FP_OFF(SourceStr), /* адрес источника */
          0xFFFF, 0x0010, /* адрес начала НМА */
          80 ); /* всего 80 байт */
/* Перенос блока информации из НМА
   в основную память и его вывод. */
movedata(0xFFFF, 0x0010, /* адрес начала НМА */
          FP_SEG(DestStr), /* адрес источника */
          FP_OFF(DestStr),
          80 ); /* всего 80 байт */
puts(DestStr);
/* Демонстрация того, что НМА распределяется
   только один раз: попытка выделить еще 20К байт
   завершается неудачей, хотя использовано только
   20К байт из общего размера 64К байт. */
if(XmsRequestHMA(20*1024))
    printf("Ошибка при распределении "\
           "20К байт НМА. Код-%x.\n"
           "Причина в том, что НМА "\
           "распределяется однажды.\n",
           XmsError);
/* Освобождение НМА. */
if(XmsReleaseHMA())
{
    printf("Ошибка при освобождении НМА. Код-%x\n",
           XmsError);
}

```



```

        if(!oldA20)
            XmsGlobalDisableA20(); /* восстановление
                                   состояния линии A20 */
        return(5);
    }
    /* Восстановление состояния линии A20. */
    if(!oldA20)
        XmsGlobalDisableA20();
    return (0); /* тест завершен */
}

```

Для получения загрузочного модуля используется файл проекта (например, L6_32.PRJ), в который включены файлы L6_11.C - L6_16.C, L6_19.C и L6_32.C. Далее приводится пример исполнения программы в случае, когда в системе имеется НМА, которая не была использована MS-DOS или другими резидентными программами:

```

D:\SOURCE\XMS>L6_32.EXE
В системе установлена расширенная (Extended) память
Версия XMS: 3.0
Обнаружена НМА (High Memory Area)
Линия A20 заблокирована. Попытка деблокирования
Тестовая строка для переноса через НМА
Ошибка при распределении 20К байт НМА. Код-91.
Причина в том, что НМА распределяется однажды.

```

В параграфе дан минимальный набор функций интерфейса прикладной программы по управлению расширенной памятью. Используя приведенные примеры Си-функций, средствами Turbo C легко построить собственную библиотеку поддержки управления отображаемой и расширенной памятью.

ЛИТЕРАТУРА

1. *Скляров В.А.* Программное и лингвистическое обеспечение персональных ЭВМ: Новые системы. — Мн.: Выш. шк., 1992.

2. *Скляров В.А.* Программное и лингвистическое обеспечение персональных ЭВМ: Системы общего назначения. — Мн.: Выш. шк., 1992.

3. IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference. — IBM, 1987.

4. *Jourdain R.* Programmer's Problem Solver for the IBM PC, XT & AT. — N.Y.: Brady Book, 1988.

5. *Schildt H. C.* Power User's Guide. — Osborn McGraw-Hill, 1988.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Адаптер асинхронной параллельной связи 213-215, 217-220

— — последовательной связи 121

Ассемблер встроенный (in-line) 5

— — в функциях-членах классов 10

AND-маска курсора "мыши" см. Курсор манипулятора "мышь"

Базовый адрес адаптера параллельной связи 217

— — — последовательной связи 127, 136

Бит тайм-аута 136

Блок памяти НМА 248, 249, 283, 284

— — — особенности использования 283, 284

— — — размещение ядра MS-DOS 249

— памяти ЕМВ 249

— памяти УМВ 249, 250

— — — размещение ядра MS-DOS 250

— параметров для переноса информации в
расширенной памяти 257

Бод 123

Буфер печати 214

Возврат значений из функции 20, 21

— — — — описанной как Interrupt 36, 37

"Вращение" регистра клавиатуры 119

Вызов ассемблерной процедуры из Си-функции 13, 26

— Си-функции из ассемблерной процедуры 26, 30

— функции с модификатором Interrupt 34, 35

"Горячая" клавиша 114

"Горячее пятно" курсора "мыши" в графическом режиме 191

Делитель частоты передачи/приема 128

Драйвер манипулятора "мышь" 182-184

— — — обработчик прерываний от адаптера 184

- Драйвер манипулятора "мышь", обработчик прерывания 10h 184
 - — — обработчик прерывания 33h см. Прерывание 33h
 - — — секция инсталляции 183
 - — — состояние по умолчанию 186
 - — — формы поставки 182

Драйвер EMM 230

- XMS 248

ЕММ-драйвер см. Драйвер EMM

ЕММ-префикс 229

EMS-память см. Память отображаемая

EMS-плата 227, 228

Завершение программы нерезидентное 31

- — резидентное 32, 53-55

Запрос-ответный режим 123

Интерфейс Си-программы с ассемблером 5-30

- RS-232 121-126

- — сигналы 123-126

- — формат слова информации 121-123

Имя в ассемблерной процедуре 17

- сегмента 16

Информация BIOSa о принтерах и других периферийных устройствах 215-217

Карта отображения 228

Каскад обработчиков прерываний 40

Код ошибки ЕММ-драйвера 232

- — XMS-драйвера 252

Командный протокол 162

Компьютер ведомый 162

- ведущий 162

- удаленный 162

Контроль по тайм-ауту принтера 214, 217

- — — адаптера последовательной связи 136

Координаты курсора "мыши" пиксельные 194

- — — — "виртуальные" 194

Курсор манипулятора "мышь" 187-189, 190-193

- — — в графическом режиме видеоадаптера 190-193

- — — — — белый непрозрачный 193

- — — — — в режиме 4 и 5 190

- — — — — видимый на любом фоне 192

- Курсор манипулятора "мышь" в текстовом режиме видеоадаптера 187-189
 - — — — — "жесткий" 187, 188
 - — — — — программируемый 187, 188
 - — — маска курсора (XOR-маска) 188, 190
 - — — маска экрана (AND-маска) 188, 190
- Линия A20 249
- "Магическое" число 128
- Манипулятор "мышь" см. Драйвер манипулятора "мышь"
- Микки 182
- Микросхема 8042 в IBM PC AT 249
- Модем 121, 122
- Настройка адаптера последовательной связи на генерацию прерываний 139
- Немодемное соединение 158-162
- Область переноса данных 79
- Обмен с диском резидентной программы 77-81
- Обработчик прерываний асинхронной связи 150-157
- Определение присутствия XMS-драйвера 251, 252
 - типа драйвера, обслуживающего EMS 230
- Ошибка кадрирования (framing error) 122
- Память отображаемая (Expanded Memory) 227-230
 - — использование резидентной программой 245
 - расширенная (Extended Memory) 247-287
 - системная 249
 - "тенивая" см. Память системная
- Передача параметров в ассемблерную процедуру 18-26
 - — в функцию, описанную с модификатором Interrupt 35-37
- Переключение стека в резидентной программе 65-68
 - PSP в резидентной программе 68-70
- Переопределение обработчика прерывания 41
- Перенос данных по интерфейсу RS-232C 131, 132
 - — — — — последовательный опрос 132
 - — — — — управляемый прерываниями 139-143
 - файлов по интерфейсу RS-232C 164-179
- Перехват прерывания 43-52
 - — критической ошибки MS-DOS 81-82
 - — по нажатию клавиши Ctrl-Break 83
- Позиционирование курсора "мыши" 201-203

Порог удвоенной скорости курсора манипулятора "мышь" 185, 196
 Порт адаптера параллельной связи см. Регистр внутренний
 — — последовательной связи см. Регистр внутренний
 Порт позиционирующего устройства 180
 Предотвращение повторной загрузки резидентной программы 55-60
 — повторного входа в MS-DOS резидентной программой 70-77
 Прерывание 08h — Системный таймер 84
 Прерывание 09h — Клавиатура 114
 Прерывание 0Bh — Адаптер COM2 в IBM AT (COM1 в IBM XT) 127
 Прерывание 0Ch — Адаптер COM1 в IBM AT (COM2 в IBM XT) 127
 Прерывание 11h — Определение состава оборудования 126
 Прерывание 13h — Управление накопителями на дисках 77
 Прерывание 14h — Управление адаптером последовательной
 связи 136-139
 — — AH = 00h — Задание формата слова информации 137
 — — AH = 01h — Запись символа 138
 — — AH = 02h — Чтение символа 138
 — — AH = 03h — Определение состояния адаптера 138
 — — AH = 04h — Расширенная инициализация 137
 — — AH = 05h — Расширенное управление модемом 138
 — — AX = 0500h — Определение состояния регистра
 управления модемом 138
 — — AX = 0501h — Установка регистра управления
 модемом 139
 Прерывание 16h — Ввод информации с клавиатуры 117
 Прерывание 17h — Управление принтером 219, 220
 — — AH = 00h — Печать символа 219
 — — AH = 01h — Инициализация адаптера и принтера 219
 — — AH = 02h — Определение состояния принтера 219
 Прерывание 1Ch — Определяемое пользователем прерывание от
 таймера 84
 Прерывание 21h — Системные функции MS-DOS
 — — AH = 21h — Установка области переноса данных 79
 — — AH = 25h — Установка вектора прерывания в ТВП 48
 — — AH = 2Fh — Определение адреса области переноса
 данных 79
 — — AH = 31h — Резидентное завершение программы 32
 — — AH = 33h — Определение и установка реакции
 на нажатие Ctrl-Break 83
 — — AH = 34h — Определение адреса флага повторной
 входимости в MS-DOS 72
 — — AH = 35h — Чтение вектора прерывания из ТВП 32
 — — AH = 49h — Освобождение блока памяти 86

- Прерывание 21h — Системные функции MS-DOS
- — AH = 50h — Установка идентификатора активной программы 69
 - — AH = 62h — Определение идентификатора активной программы 69
- Прерывание 23h — Обработка нажатия клавиш Ctrl-Break 83
- Прерывание 24h — Обработка критической ошибки 81
- Прерывание 27h — Резидентное завершение программы 32
- Прерывание 28h — Безопасная активизация резидентных программ 71
- Прерывание 33h — API-интерфейс с манипулятором "мышь" 185-202
- — AX = 0000h — Определение состояния оборудования 186
 - — AX = 0001h — Включение курсора "мыши" на экране 197
 - — AX = 0002h — Выключение курсора "мыши" на экране 197
 - — AX = 0003h — Определение позиции курсора и состояния кнопок 198
 - — AX = 0004h — Перемещение курсора "мыши" 202
 - — AX = 0005h — Определение количества нажатий кнопок 199
 - — AX = 0006h — Определение количества отпусканий кнопок 199
 - — AX = 0007h — Установка диапазона перемещения курсора по горизонтали 194
 - — AX = 0008h — Установка диапазона перемещения курсора по вертикали 194
 - — AX = 0009h — Установка формы курсора в графическом режиме 191
 - — AX = 000Ah — Установка формы курсора в текстовом режиме 188
 - — AX = 000Bh — Определение значения счетчиков сигналов мышки 199
 - — AX = 000Fh — Установка чувствительности курсора 195
 - — AX = 0013h — Установка порога удвоенной скорости курсора 196
 - — AX = 001Bh — Определение чувствительности курсора 196
- Прерывание 2Fh — Мультиплексное прерывание 59, 251
- — AH=43h — Состояние XMS-драйвера 251
 - — AX=4300h — Определение присутствия XMS-драйвера 251
 - — AX=4310h — Определение точки входа в XMS-драйвер 251
- Прерывание 67h — API-интерфейс с EMS-памятью 230-247
- — AH = 40h — Определение состояния EMS 231
 - — AH = 41h — Определение сегмента начала физического фрейма EMS 231
 - — AH = 42h — Определение общего объема EMS-памяти 231

- Прерывание 67h — API-интерфейс с EMS-памятью
 - — AH = 43h — Распределение логических страниц EMS-памяти 232
 - — AH = 44h — Построение карты отображения 232
 - — AH = 45h — Закрытие EMM-префикса 232
 - — AH = 46h — Определение версии EMS 233
 - — AH = 47h — Сохранение текущей карты отображения 233
 - — AH = 48h — Восстановление текущей карты отображения 233

Принтер параллельный 213

- состояние по умолчанию 215

Программа реентерабельная 31

- резидентная 31
- — инициализирующая часть 32, 43-64
- — резидентная часть 32, 65-89
- фоновая 65

Программное подтверждение приема информации 160, 162-164

Протокол XON/XOFF 126

Радиомодем 122

Распределение адресов памяти в IBM PC 228

- — — — — на основе процессоров 80286/386 248

Регистр внутренний адаптера параллельной связи 217, 218

- — — — — регистр данных 217
- — — — — регистр состояния 217, 218
- — — — — регистр управления 217, 218

Регистр внутренний адаптера последовательной связи 126-135

- — — — — делитель частоты 128, 129
- — — — — регистр данных передатчика 127
- — — — — регистр данных приемника 127
- — — — — регистр идентификации прерывания 134, 135
- — — — — регистр разрешения прерываний 133
- — — — — регистр состояния линии 131, 132
- — — — — регистр состояния модема 131
- — — — — регистр управления линией 129
- — — — — регистр управления модемом 135

Регистр сдвиговый 122

Режим работы манипулятора "мышь" 181

- — — — — программное переключение (softswitch) 181
- — — — — Microsoft Mode 181
- — — — — Mouse System Mode 181

"Рукопожатие" (handshaking) см. Запрос-ответный режим

Своппинг 80

Семафор бинарный 68
 Си-порядок именованя и передачи параметров 24
 Слово подключенного оборудования 215
 Старт-бит 122
 Старт-стопный метод 122
 Стек в момент входа в функцию 19
 — — — — — описанную с модификатором interrupt 36
 Стоп-бит 122
 Страница EMS логическая 228
 — — физическая 228
 Строка модема управляющая 125
 Схема построения манипулятора "мышь" 181
 "Сценарий" 118

 Требования к обработчику прерывания 33, 65-89

 Удаление резидентной программы из памяти 85-89
 Условие BREAK 126

 "Фильтр" 57
 Флаг повторной входимости в MS-DOS 72
 Формат слова информации интерфейса RS-232 121-123
 — — — — — при инициализации через BIOS 137
 Формирование курсора "мыши" в графическом режиме 190
 Фрейм стека 18
 — страниц 228
 Функции библиотеки Turbo C для включения машинного кода 9
 — интерфейса с манипулятором "мышь" см. Прерывание 33h
 — — с EMS-памятью см. Прерывание 67h
 — — — — версии EMS 3.2 230-233
 — — — — версии EMS 4.0 233
 Функции интерфейса с XMS-памятью 230-247
 — — — — Определение присутствия XMS-драйвера 251, 252
 — — — — AH=00h — Определение номера версии XMS 255
 — — — — AH=01h — Распределение HMA 255
 — — — — AH=02h — Освобождение HMA 255
 — — — — AH=03h — Глобальное деблокирование линии A20 255
 — — — — AH=04h — Глобальное блокирование линии A20 255
 — — — — AH=05h — Локальное деблокирование линии A20 255
 — — — — AH=06h — Локальное блокирование линии A20 256
 — — — — AH=07h — Запрос состояния линии A20 256
 — — — — AH=08h — Запрос состояния расширенной памяти 256
 — — — — AH=09h — Распределение блока расширенной
 памяти EMB 256

Функции интерфейса с XMS-памятью

- — — — AH=0Ah — Освобождение блока расширенной памяти EMB 256
- — — — AH=0Bh — Перенос блока информации в расширенную память EMB 257
- — — — AH=0Ch — “Запирание” блока расширенной памяти EMB 257
- — — — AH=0Dh — “Отпирание” блока расширенной памяти EMB 258
- — — — AH=0Eh — Возврат информации о XMS-префиксе 258
- — — — AH=0Fh — Перераспределение блока расширенной памяти EMB 258
- — — — AH=10h — Распределение блока памяти UMB 258
- — — — AH=11h — Освобождение блока памяти UMB 258

Функции с модификатором interrupt 34

XMS-драйвер см. Драйвер XMS

XMS-память см. Память расширенная

XMS-префикс 256

XOR-маска курсора “мыши” см. Курсор манипулятора “мышь”

Чтение позиции курсора и состояния кнопок “мыши” 198-201

“Чувствительность” курсора манипулятора “мышь” 185, 195

“Эхо”-возврат 57

“Эхо”-константа 57

УКАЗАТЕЛЬ ЛЕКСЕМ

asm	5	/HMAMAIN=	252
AUX	126	interrupt	34
BETTA.EXE	220	keep()	53
bioscom()	139	LPT1	213
cdecl	24	LPT2	213
.CODE	14	LPT3	213
COM1	126	.MODEL	14
COM2	126	/NUMHANDLES=	252
COM3	126	pascal	24
COM4	126	#pragma inline	8
.DATA	14	PRN	224
.DATA?	14	PUBLIC	17
DOS=HIGH	249	QEMM.SYS	247
DOS=HIGH,UMB	250	setcbrk()	83
__emit__()	9	setdta()	79
EMM.LIB	250	setvect()	48
EMM.SYS	230	_stklen	55
EMM386.EXE	247	this	10
EXTRN	16	VDISK.SYS	251
freemem()	86	XMS.H	253
getcbrk()	83	XMS2EMS.SYS	230
getdta()	79		
getvect()	48		
_heaplen	55		
HIMEM.SYS	248		

СПИСОК УСЛОВНЫХ ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

НГМД	- накопитель на гибких магнитных дисках
ОП	- оперативная память
ОС	- операционная система
ПЗУ	- постоянное запоминающее устройство
ППИ	- программируемый периферийный интерфейс
ПЭВМ	- персональная электронно-вычислительная машина
ТВП	- таблица векторов прерывания
ЭЛТ	- электронно-лучевая трубка
АН	- регистр общего назначения процессора Intel 8086
AL	- регистр общего назначения процессора Intel 8086
API	- Application Program Interface - интерфейс прикладной программы
AX	- регистр общего назначения процессора Intel 8086
ВН	- регистр общего назначения процессора Intel 8086
BIOS	- Base Input-Output System - базовая система ввода-вывода
BL	- регистр общего назначения процессора Intel 8086
BP	- Base Pointer - регистр-указатель базы процессора Intel 8086
BX	- регистр общего назначения процессора Intel 8086
CF	- Carry Flag - флаг переноса процессора Intel 8086
CGA	- Color Graphics Adapter - цветной графический адаптер
CH	- регистр общего назначения процессора Intel 8086
CL	- регистр общего назначения процессора Intel 8086
CRT	- Cathode Ray Tube - электронно-лучевая трубка
CRTC	- Cathode Ray Tube Controller - контроллер электронно-лучевой трубки
CS	- сегментный регистр кода процессора Intel 8086
CTS	- Clear To Send - линия интерфейса RS-232C "Готовность модема"
CX	- регистр общего назначения процессора Intel 8086
DCD	- Data Carrier Detected - линия интерфейса RS-232C "Соединение с удаленным модемом"
DH	- регистр общего назначения процессора Intel 8086
DI	- Destination Index - индекс-регистр назначения процессора Intel 8086
DL	- регистр общего назначения процессора Intel 8086

DS - сегментный регистр данных процессора Intel 8086
 DSR - Data Set Ready - линия интерфейса RS-232C
 "Готовность набора данных"
 DTA - Data Transfer Area - область переноса данных диска
 DTR - Data Terminal Ready - линия интерфейса RS-232C
 "Готовность терминала данных"
 DX - регистр общего назначения процессора Intel 8086
 EGA - Enhanced Graphics Adapter - усовершенствованный
 графический адаптер
 EMB - Extended Memory Block - блок памяти EMB
 EMM - Expanded Memory Manager - драйвер управления
 EMS-памятью
 EMS - Expanded Memory Specification - спецификация
 управления отображаемой памятью
 EOI - End Of Interrupt - команда завершения
 обслуживания аппаратного прерывания
 ES - дополнительный сегментный регистр процессора
 Intel 8086
 GND - Ground - линия интерфейса RS-232C "Логический ноль"
 HGC - Hercules Graphics Card - монохроматический
 адаптер Hercules
 HMA - High Memory Area - первые 64K байт расширенной памяти
 IP - Instruction Pointer - регистр-указатель исполняемой
 инструкции процессора Intel 8086
 IRQ - Interrupt Request - запрос прерывания
 ISR - Interrupt Service Routine - программа обработки
 прерывания
 LIM EMS - спецификация управления отображаемой памятью,
 разработанная компаниями Lotus, Intel и Microsoft.
 MCGA - Multi Color Graphics Array Adapter - адаптер
 "многоцветный массив"
 PIC - Programmable Interrupt Controller -
 программируемый контроллер прерываний
 PID - Program Identifier - идентификатор программы
 PSP - Program Segment Prefix - префикс сегмента программы
 RI - Ring Indicator - линия интерфейса RS-232C
 "Индикатор вызова"
 RTS - Request To Send - линия интерфейса RS-232C
 "Запрос на передачу"
 RXD - линия интерфейса RS-232C приема данных из модема
 SI - Source Index - индекс-регистр источника
 процессора Intel 8086
 SP - Stack Pointer - регистр-указатель текущей
 вершины стека процессора Intel 8086
 SS - сегментный регистр стека процессора Intel 8086
 TSR - Terminate and Stay Resident - резидентная программа
 TXD - линия интерфейса RS-232C передачи данных в модем
 UMB - Upper Memory Block - блок памяти UMB
 VGA - Video Graphics Array -адаптер "графический
 видеомассив"
 XMS - Extended Memory Specification - спецификация
 управления расширенной памятью

ОГЛАВЛЕНИЕ

Предисловие	3
1. Интерфейс Си-программы с ассемблером	5
1.1. Включение ассемблерных фрагментов в Си-функцию	5
1.2. Вызов ассемблерной процедуры из Си-функции	13
1.3. Вызов Си-функции из ассемблерной процедуры	26
2. Разработка резидентных программ	31
2.1. Введение в резидентные программы	31
2.2. Функции, выполняемые инициализирующей частью TSR-программы	43
2.3. Построение резидентной части TSR	65
2.4. Удаление резидентной программы из памяти	85
2.5. Резидентная программа записи текстового окна в файл	89
2.6. Резидентные программы, использующие прерывание от клавиатуры. "Горячие" клавиши	114
2.7. Резидентные программы, использующие прерывание 16h	117
3. Управление адаптером асинхронной последовательной связи	121
3.1. Аппаратные средства асинхронной последовательной свя- зи. Интерфейс RS-232	121
3.2. Средства BIOSa управления асинхронной связью.	135
3.3. Управляемая прерываниями асинхронная связь	139
3.4. Организация связи между компьютерами по интерфейсу RS-232	158
4. Ввод информации с манипулятора "мышь"	180
4.1. Аппаратные средства манипулятора "мышь"	180
4.2. Программная поддержка манипулятора. Основные компо- ненты драйвера "мыши"	182
4.3. Основные функции интерфейса прикладной программы с манипулятором "мышь"	185
5. Управление адаптером параллельной связи и принтером	213
5.1. Технические и программные средства управления адап- тером параллельной связи и принтером	213
5.2. Информация BIOSa о принтерах и других периферийных устройствах	215
5.3. Порты адаптера параллельной связи.	217
5.4. Средства BIOSa управления адаптером параллельной связи	219
5.5. Инициализация адаптера параллельной связи и принтера	220
5.6. Передача символов	223
6. Управление отображаемой и расширенной памятью	227
6.1. Отображаемая память EMS	227
6.2. Управление расширенной памятью.	247
Литература.	288
Предметный указатель	289
Указатель лексем	297
Список условных обозначений и сокращений	298

Учебное издание

Касаткин Александр Иванович

**ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ СИ**

Системное программирование

**Заведующий редакцией Л.Д. Духвалов
Редактор М.С. Молчанова**

ИБ N 3381

Подписано в печать с авторского оригинала-макета 22.02.93. Формат 84x108/32. Бумага кн.-журн. Гарнитура Таймс. Офсетная печать. Усл.печ. л. 15,96. Усл.кр.-отт. 16,38. Уч.-изд.л. 16,91. Тираж 45 000 экз. Заказ 162.

Издательство "Вышэйшая школа" Министерства информации Республики Беларусь. 220048, Минск, проспект Машерова, 11.

Типография издательства "Беларускі Дом друку". 220041, Минск, проспект Ф. Скорины, 79.



Уважаемый читатель !

Вы приобрели серию из трех книг “Профессиональное программирование на языке Си” и собираетесь изучать их, расположившись удобно за компьютером и проверяя теорию практикой.

Мы поможем Вам извлечь максимальную пользу из Вашей покупки, освободив от рутинного утомительного труда по набору текстов программ и подготовке файлов проектов.

Спецфакультет МРТИ, научно-производственно-коммерческая фирма “ТЕХИНКОМ” и Автор подготовили для Вас компаньон-дискету с избавленными от опечаток текстами всех примеров, приведенных в трех книгах.

Для получения дискеты с примерами необходимо прислать заявку по адресу: 220071, Республика Беларусь, г.Минск, ул.Платонова, 39, МРТИ, корп.5, спецфакультет, комн. 806, 322, или позвонить по телефону (0172)31-55-88, 39-86-32. В заявке укажите Ваш адрес и телефон.

Дискеты высылаются после предоплаты. Условия предоплаты и стоимость заказа (стоимость программ, дискеты и почтовые услуги) сообщаются после получения заявки.

Вы можете также приобрести у нас любую из трех книг комплекса.

Принимаются заявки на оптовую поставку книг и компаньон-дискет.

Вы хотите приобрести новую специальность, получить знания в области новейших информационных технологий, изучить имеющееся и разрабатывать собственное программное обеспечение, создавать и эксплуатировать базы данных, изучить аппаратные средства, научиться ремонтировать и обслуживать персональные компьютеры!

Все это и многое другое Вы найдете на

Специальном Факультете
Переподготовки Кадров
МИНСКОГО РАДИОТЕХНИЧЕСКОГО
ИНСТИТУТА.

С П Е Ц Ф А К У Л Ь Т Е Т

Вам предлагает:

Переподготовку с выдачей диплома о присвоении новой специальности и квалификации инженер-системотехник по специальностям:

• **МИКРОПРОЦЕССОРНЫЕ СИСТЕМЫ**, специализации:

— “Проектирование и применение микропроцессорных систем”;

— “Управляющие ЭВМ”;

• **ПЕРСОНАЛЬНЫЕ ПРОФЕССИОНАЛЬНЫЕ ЭВМ**, специализации:

— “Применение ПП ЭВМ”;

— “Ремонт и техническое обслуживание ПП ЭВМ”;

— “Локальные сети ЭВМ”.

Продолжительность обучения — 6, 9, 18 месяцев (с отрывом и без);

Переподготовку с выдачей удостоверения (повышение квалификации) по направлениям:

— ПРИМЕНЕНИЕ ПЕРСОНАЛЬНЫХ ПРОФЕССИОНАЛЬНЫХ ЭВМ;

— РЕМОНТ И ТЕХНИЧЕСКОЕ ОБСЛУЖИВАНИЕ ПП ЭВМ;

— МИКРОПРОЦЕССОРНЫЕ СИСТЕМЫ И КОМПЛЕКТЫ МП БИС;

— ЛОКАЛЬНЫЕ СЕТИ.

Продолжительность обучения — 1, 2 месяца;

СЕРТИФИЦИРОВАННЫЙ ЦЕНТР ОБУЧЕНИЯ

На факультете функционирует первый в Республике Беларусь официальный центр обучения фирмы COMPUTER ASSOCIATES INTERNATIONAL (USA), который проводит обучение по следующим специально разработанным программам:

- Система управления базами данных Clipper Summer'87;
- Система управления базами данных Clipper 5.01;
- Электронная таблица SuperCalc;
- Программирование в BORLAND C++.

Слушателям, успешно сдавшим выпускные экзамены по программам Clipper или SuperCalc, выдается сертификат фирмы COMPUTER ASSOCIATES INTERNATIONAL INC.(USA). Вместе с сертификатом по SuperCalc слушатели получают право на приобретение по льготной цене (60% от рыночной) лицензионного русифицированного пакета SuperCalc 5.0.

Продолжительность обучения — 2 недели.

ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ

Формы обучения: дневная, вечерняя, заочная. Начало занятий — по мере комплектования групп. Каждому слушателю предоставляется возможность индивидуальной работы за компьютером в течение рабочего дня. Иногородние обеспечиваются общежитием гостиничного типа. Оплата за наличный и безналичный расчет. Для поступления необходимо прислать заявление на имя декана и направление-гарантийное письмо (для обучающихся от предприятий). После этого высылаются договор на обучение и перечень остальных документов. Заявки принимаются на протяжении учебного года.

ПОЛЬЗОВАТЕЛЯМ Nowell NetWare!

Лаборатория локальных сетей SFLanLabs спецфакультета предлагает оригинальную систему электронной почты, BLUE BOX, позволяющую расширить возможности пользователей сети NNW. Гарантируются обязательность вручения с уведомлением отправителя, контроль времени ответа, конфиденциальность переписки. Незаменима для руководителя.

Принимаем заявки на разработку программных продуктов и баз данных, установку сетевого ПО.

НАШ АДРЕС

220071, г.Минск, ул. Платонова, 39, МРТИ,
корп. 5, спецфакультет, комн. 322 , 806.

Телефоны: (0172) 31-55-88, 39-86-03, 39-86-32.

ИСПОЛЬЗУЙТЕ ВАШ ШАНС !

A. N. Kacatikhin